Kim Kraft

Tekstiviestirajapinnan kehitys web-pohjaiseen verkkotietojärjestelmään

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tietotekniikka
Insinöörityö
10.02.2013

Metropolia

Tiivistelmä

Tekijä(t) Otsikko Sivumäärä Aika	Kim Kraft Tekstiviestirajapinnan kehitys web-pohjaiseen verkkotietojärjestelmään 46 sivua + 2 liitettä 10.02.2013
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Tietoverkot
Ohjaaja(t)	Yliopettaja Janne Salonen

Työssä tehtiin tekstiviestirajapinta osaksi Keypro Oy:n vuonna 2009 kehittämää KeyCore-sovelluskehystä. Tarkoituksena oli tarjota rajapinta tekstiviestin lähetykseen kaikille KeyCoreen perustuville tuotteille. Työ tehtiin käyttäen Python-ohjelmointikieltä ja Django-sovelluskehystä.

Aluksi työssä käsitellään yleisesti Keypro Oy:n verkkotietojärjestelmää, jotta lukija saa pienen käsityksen järjestelmästä, johon tekstiviestirajapinta rakennettiin. Sitten kerrotaan jonkin verran Pythonista ohjelmointikielenä ja sen eduista muun muassa syntaksin puolesta. Django-sovelluskehys sai ihan oman kappaleen, koska kyseessä on todella laaja sovelluskehys. Djangon toiminnallisuutta esittelen luvussa luomalla esimerkkinä yksinkertaisen Django-sovelluksen. Tekstiviestirajapinta on Django-sovellus, joten Djangon perusteiden läpikäyminen auttaa myöhemmin tekstiviestirajapinnan ymmärtämistä.

Tekstiviestirajapinnan kehitys noudattaa perinteistä kehityskaavaa, joka on määrittely, suunnittelu, toteutus ja testaus. Kyseisessä järjestyksessä opinnäytetyökin etenee. Määrittelyvaiheessa käydään lyhyesti läpi, mistä lähtökohdista lähdettiin tekemään tekstiviestirajapintaa. Määrittelystä saatiin aikaiseksi tarkka suunnitelma, jonka pohjalta lähdetään toteuttamaan rajapintaa. Toteutuskappaleessa edetään suunnitelman mukaisesti ja käytetään kuvia osasta lähdekoodia selventämään rajapinnan ohjelmointiosuuksia. Lopuksi käydään läpi testauksen osuutta toteutuksessa ja pohdiskellaan yleisesti rajapinnan käyttökohteita ja jatkokehitystä.

Tekstiviestirajapinnalle oli kysyntää asiakkaiden puolelta jo valmiiksi, joten rajapinnan tekeminen oli vain ajankysymys. Lopputulos oli onnistunut ja kaikki KeyCoreen perustuvat tuotteet voivat nyt lähettää tekstiviestejä niin halutessaan.

Avainsanat	Django, Python, tekstiviestirajapinta, verkkotietojärjestelmä

Metropolia

Abstract

Author(s) Title Number of Pages Date	Kim Kraft SMS interface for web-based GIS 46 pages + 2 appendices 10 December 2013
Degree	Bachelor of Engineering
Degree Programme	Degree programme in Information Technology
Specialisation option	Data Networks
Instructor(s)	Janne Salonen, principal lecturer

The objective of this thesis was to create an interface for text messaging (SMS) as a part of KeyCore-framework developed by Keypro Ltd and the purpose was to offer possibility to send SMSs for all KeyCore-based products.

In the beginning, thesis covers the basics of Keypro Ltd.'s geographical information system (GIS) so that the reader gets outlook of the system that this SMS-interface was built on. After that we briefly cover some basics of the Python language and for example its benefits syntax wise. Django-framework got its very own chapter since it's a very comprehensive framework and to help to explain the functionality of the framework. I will create a small Django application as the chapter progresses. Since the SMS interface is a Django application, it helps the reader later on to understand the implementation part of the SMS interface.

The development of the SMS interface follows the traditional development pattern: specification, planning, implementation and testing, in which order the thesis progresses as well. On the specification section we briefly cover the starting point of the whole development. An accurate plan was created out of the specification, which was used as a basis for the implementation section. In the implementation section, we follow the plan and use pictures of parts of the source code to clarify the programming parts. In the end, we will handle the testing section and how the testing was done simultaneously with the implementation. Also there will be general brain storming for SMS interface use cases and future development.

There was existing demand for SMS interface in KeyCore-based products, so implementation of a SMS interface was just a matter of time. The final result was successful and now all KeyCore-based products have the functionality of sending text messages.

Keywords	Python, Django, Web-based GIS, SMS interface

Sisällys

Käsitteet ja lyhenteet

Sovelluskehys	Ohjelmointikielen kirjasto, joka tarjoaa valmiita funktioita eri toimintoihin.
Django	Sovelluskehys web-ohjelmointiin Pythonilla.
Web etuliite	Tarkoitetaan internetin käytettävää sovellusta.
KeyCore	Keypron oma web-sovelluskehys, jonka päälle rakentuvat sen kaikki verkkotietojärjestelmät.
KeyCom	Yksi Keypron useista KeyCoren perustuvista tuotteista. Tarkoitettu johtoverkkojen suunnitteluun yms.
Modulaarisuus	Sovelluksen paloitteleminen pieniin toiminnallisiin osiin.
Rajapinta	Määritelmä, jonka kautta ulkopuoliset sovellukset voivat tehdä jotain tiettyjä toimintoja.
Autentikointi	Käyttäjän varmentaminen.
SMS-yhdyskäytävä	Kolmannen osapuolen palvelu joka vastaanottaa tekstiviesti parametrit internetin yli ja lähettää ne eteenpäin operaattorille, joka lähettää tekstiviestin käyttäen matkapuhelinverkkoa.
Funktio/metodi	Tarkoittavat samaa asiaa. Funktiolla ei ole omistavaa luokkaa, kuten metodilla. Metodi mm. saa parametrina instanssin sen omistavasta luokasta, jonka kautta se pystyy käsittelemään luokkamuuttujia. [34.]
Argumentti/parametri	Tarkoittavat samaa asiaa ja käsitteiden ero riippuu näkökulmasta. Funktio kutsussa annetaan argumentti, kun taas funktio käyttää sisällään parametreja. [33.]

1 Johdanto

Keypro Oy tarjoaa asiakkailleen paikkatietopalveluja, joihin kuuluvat muun muassa web-pohjaiset verkkotietojärjestelmät, jotka on tarkoitettu erilaisten verkostojen suunnitteluun ja ylläpitoon suoraan internetselaimesta käsin. Tämä opinnäytetyö tehtiin Keypro Oy:lle ja sen tavoitteena oli luoda tekstiviestirajapinta heidän verkkotietojärjestelmään.

Tekstiviestit ovat itsessään arkipäivää, jokaisen matkapuhelimesta tällainen ominaisuus löytyy, joten tekstiviestit ovat yksinkertaisuudessaan erinomainen tapa tavoittaa kuka vain ja tällaisen ominaisuuden löytyminen web-pohjaisesta verkkotietojärjestelmästä voi osoittautua hyvin käytännölliseksi. Keypro ottaa asiakkaansa mukaan yhteiskehitysryhmiin, joilta on tullut toiveita tekstiviestitoiminnolle, joten rajapinnalle oli valmiiksi kysyntää. Keypron verkkotietojärjestelmää käyttää muun muassa usea vesiosuuskunta, jos esimerkiksi juomavesi pääsee syystä tai toisesta saastumaan, niin varoituksen lähettäminen tekstiviestillä saavuttaa hetkessä suuren määrän asiakkaita. Asiakkaille voi myös lähettää tiedon mahdollisista kaivuoperaatioista tietyllä alueella tai Keypro voi vahvistaa verkkotietojärjestelmänsä tietoturvallisuutta tekstiviestiin perustuvan autentikoinnin kautta. [14; 31.]

Keypron verkkotietojärjestelmä on web-sovellus, joten se koostuu web-sovellukselle tyypillisistä loogisista osista eli palvelinpuolesta, asiakaspuolesta ja tietokannasta. Verkkotietojärjestelmän palvelinpuoli on rakennettu käyttäen Django-sovelluskehystä, joka tarkoitettu web-ohjelmointiin Pythonilla ja asiakaspuoli on tehty JavaScriptllä. Ohjelmistorakenteeltaan verkkotietojärjestelmä koostuu kahdesta osasta eli ytimestä KeyCore ja KeyCoren päälle rakennetusta tuotteesta, kuten esimerkiksi KeyComista, joka on verkkotietojärjestelmä.

Tekstiviestirajapinta tehtiin palvelinpuolelle osaksi KeyCorea, joten asiakaspuoli eli visuaalinen osa jäi tämän opinnäytetyön ulkopuolelle, joten esimerkiksi JavaScriptiä ei tässä työssä käsitellä. Asiakaspuoli kuitenkin tehtiin muiden toimesta Keypro:lla ja heidän lopputuloksensa esitellään lyhyesti. Rajapinta myös tallentaa lähetetyt tekstiviestit tietokantaan laskutusta varten. Rajapinnan kehityksen vaiheet käydään työssä läpi yksityiskohtaisesti ensin määrittelystä suunnitelmaan ja sitten toteutuksesta

testaukseen, jonka jälkeen käsitellään rajapinnan mahdollisia käyttökohteita ja pohdiskellaan jatkokehitystä.

2 Keypron verkkotietojärjestelmä

2.1 Palvelin- ja asiakaspuoli web-sovelluksessa

Ennen kuin pureudutaan verkkotietojärjestelmään, niin on hyvä käydä läpi otsikossa esiintyvät käsitteet, koska niitä tullaan käyttämään säännöllisesti tässä raportissa. Web-sovelluksesta puhuttaessa yleensä tarkoitetaan täysin selaimessa toimivaa sovellusta. Sillä voidaan korvata työpöytäsovelluksia, käyttäjän koneelle ei tarvitse olla asennettuna muuta kuin selain, jonka kautta palvelimella olevaa sovellusta käytetään internetin tai lähiverkon yli, kun taas perinteinen työpöytäsovellus täytyy olla asennettuna käyttäjän koneelle. Työpöytäsovellukset ovat toimiva ratkaisu, mutta web-pohjaisuus tuo mukanaan huomattavia etuja. [29.]

Suoritin ensimmäisen osan työharjoittelustani IT-yrityksessä, jolle muut yritykset ulkoistivat heidän tietotekniset asiat, kuten työasemien huollot, niiden ylläpidon ja ylipäätänsä kaikki hommat, jotka liittyivät tietotekniikkaan jollain tavalla. Itse työskentelin asennuspuolella, jonne tulivat asiakkaiden hajonneet tai uudet tietokoneet, jotka sitten korjasimme tai esiasensimme käyttövalmiiksi. Jokaiselle meille saapuneelle uudelle työasemalle asensimme Windowsin lisäksi yrityskohtaisesti erilaisia sovelluksia, joista jotkut oli tehty täysin yrityksen omaan käyttötarkoitukseen, kuten kasa erilaisia laskutus-, myynti- ja suunnitteluohjelmia ja tähän päälle vielä yritysten työntekijöiden henkilökohtaisia toivomuksia. Yritykset tietenkin halusivat, että ohjelmat on päivitetty aina uusimpaan versioon, mutta kun yrityksiä on asiakkaana yli 20 ja kaikilla on eri ohjelmia käytössä, eikä läheskään kaikkia tarvittavia ohjelmia löydy internetistä, vaan yrityksen omalta palvelimelta, niin yhden työaseman asennusaika voi kasvaa suureksi. Sitä voi verrata tilanteeseen, jossa uudelle työasemalle tarvitsisi asentaa lukuisten sovellusten sijaan vain yksi eli selain.

Kehittämisen näkökulmasta web-sovellus on hieno asia, koska työpöytäsovelluksia kehittäessä täytyy ottaa huomioon eri käyttöjärjestelmät, jos haluaa tarjota sovelluksen mahdollisimman suurelle käyttäjäkunnalle, esimerkiksi Windows-sovelluksen

muuntaminen Linuxille ei ole yhden illan juttu. Web-sovellusten tekijöiden ei tarvitse tästä huolehtia, koska sovellusta käytetään selaimen kautta, mutta selaimissakin on eroja. Parhaimmassa tapauksessa selain erot voi ratkaista muutamalla lisäkoodirivillä. Kehityksen lisäksi ylläpito helpottuisi huomattavasti, koska web-pohjaisessa ratkaisussa sovellukset asennetaan yhteen sijaintiin palvelimelle, jolloin hallittavana on vain yksi sovellus lukuisten paikallisten kopioiden sijaan. Kun sovellukset ovat yhdessä paikassa, niin ne on helppo pitää ajan tasalla, jolloin työntekijöillä on aina käytössä uusin versio. Myös työaseman hajotessa äkillisesti työntekijä ei menettäisi monien päivien työtä, koska kaikki on tallennettu tietokantaan palvelimelle, jota hallinnoi IT-alan ammattilaiset, eikä sen täten pitäisi olla niin altis tiedon menetykselle. Tämä säästäisi asentajien ja yritysten työntekijöiden aikaa, kun hajonneelta kovalevyltä ei tarvitse yrittää palautella viikon työpanosta. [30.]

Yleensä web-sovellus perustuu kolmeen käsitteeseen, jotka ovat asiakas, palvelin ja tietokanta. Web-sovelluksen asiakaspuolella tarkoitetaan usein selainta, jota loppukäyttäjä käyttää. Asiakaspuoli keskustelee verkon yli palvelimen eli palvelinpuolen kanssa, joka taas yleensä keskustelee tietokannan kanssa. Palvelinpuolella voi usein olla useita palvelimia, mutta web-palvelin on se, jonka kanssa asiakaspuoli keskustelee. Web-palvelin voi sitten keskustella edelleen tietokantapalvelimen kanssa ja jopa erillisen sovelluspalvelimen kanssa, mutta palvelinpuolen rakenteesta riippumatta web-pohjainen toimintaperiaate pysyy samana. [29.]

Web-sovelluksessa on vielä haittapuolia niitä verrattaessa vastaaviin työpöytä-sovelluksiin. Esimerkkinä voi mainita turvallisuuden, kun käytetään julkista internetiä arkaluontoisen tiedon siirtoon, niin riskit ovat suurempia. Kuitenkin suurin osa turvallisuusongelmista johtuu sovelluksen huolimattomasta ohjelmoinnista, joka mahdollistaa sen hyväksikäytön esimerkiksi XSS-tietoturva-aukon kautta. Suuret web-sovellukset eivät välttämättä toimi yhtä jouhevasti kuin vastaavat työpöytäsovellukset, koska selainteknologia ei ole vielä niin kehittynyttä ja web-sovellukset toimivat korkeimmalla OSI-mallin tasolla eli sovelluksella on enemmän vaiheita suoritettavana operaatioissaan. Tulevaisuudessa nämä haittapuolet tullaan kuitenkin kuromaan umpeen. [20; 21.]

2.2 Verkkotietojärjestelmä yleisesti

Paikkatietojärjestelmä on nimensä mukaisesti järjestelmä, jossa kohteiden sijainti on olennainen osa tietosisältöä. Perinteisesti paikkatietojärjestelmä on ollut työasemalle asennettava erillinen ohjelmisto. Paikkatietojärjestelmä koostuu yleensä tietokannasta, johon käsiteltävä tieto tallennetaan käyttöliittymästä, joka on tyypillisesti graafinen, ja analyysityökaluista, jotka analysoivat paikkatietoa eri tavoilla. Paikkatieto terminä viittaa kaikenlaiseen paikkatietoon, kun Keypron kohdalla paikkatieto on tietoa verkostoista, niin on luontevaa kutsua heidän järjestelmänsä verkkotietojärjestelmäksi. [3.]

Keyprolla nähtiin potentiaali web-pohjaisessa ratkaisussa vuonna 2009. He alkoivat edelläkävijöinä kehittää web-pohjaista ja avoimeen standardiin perustuvaa verkkotietojärjestelmää. Keypron verkkotietojärjestelmä on täysiverinen web-sovellus ja karkeasti verrattavissa esimerkiksi Google Mapsiin, jonka päälle on lisätty suunnittelutyökaluja sijainti- ja ominaisuustiedon tuottamiseen ja analysointiin. Verkkotietojärjestelmä on tarkoitettu erinäisten verkostojen suunnitteluun ja paikantamiseen, johon se tarjoaa laajan valikoiman erilaisia työkaluja. Olkoon nämä verkostot sitten vesijohtoa, kaapelia tai katuvalaistusta, niin jokaista tarkoitusta varten Keypro on räätälöinyt oman verkkotietojärjestelmän. Keypron verkkotietojärjestelmä koostuu kahdesta osasta eli tuotteesta, kuten televerkkojen suunnitteluun ja dokumentointiin tarkoitettu KeyCom tai vesi- ja viemäriverkoille tarkoitettu KeyAqua, jotka tarjoavat toimialan erityistoiminnot ja kaikille toimialoille yhteiset toiminnot tarjoavasta KeyCoresta. [1.]

Verkkotietojärjestelmä vaatii tietenkin tietokannan, jotta siihen voi tallentaa suunnittelutöitä, paikkatietoa ja ylipäätänsä pysyvää tietoa.

2.3 KeyCore

KeyCore on Keypron vuonna 2009 Pythonilla ja useiden Python-sovelluskehysten avulla ohjelmoitu sovelluskehys, joka on ydinosa Keypron uusissa web-pohjaisissa verkkotietojärjestelmissä. KeyCore tarjoaa rajapintoja yleisiin käyttötarkoituksiin ja hoitaa vaadittuja perustehtäviä, jotka ovat yhteisiä kaikille tuotteille, jolloin esimerkiksi

samoja toiminnallisuuksia ei tarvitse ohjelmoida jokaiseen tuotteeseen erikseen, jolloin myös verkkotietojärjestelmän hallinnointi helpottuu. [2, s. 3.]

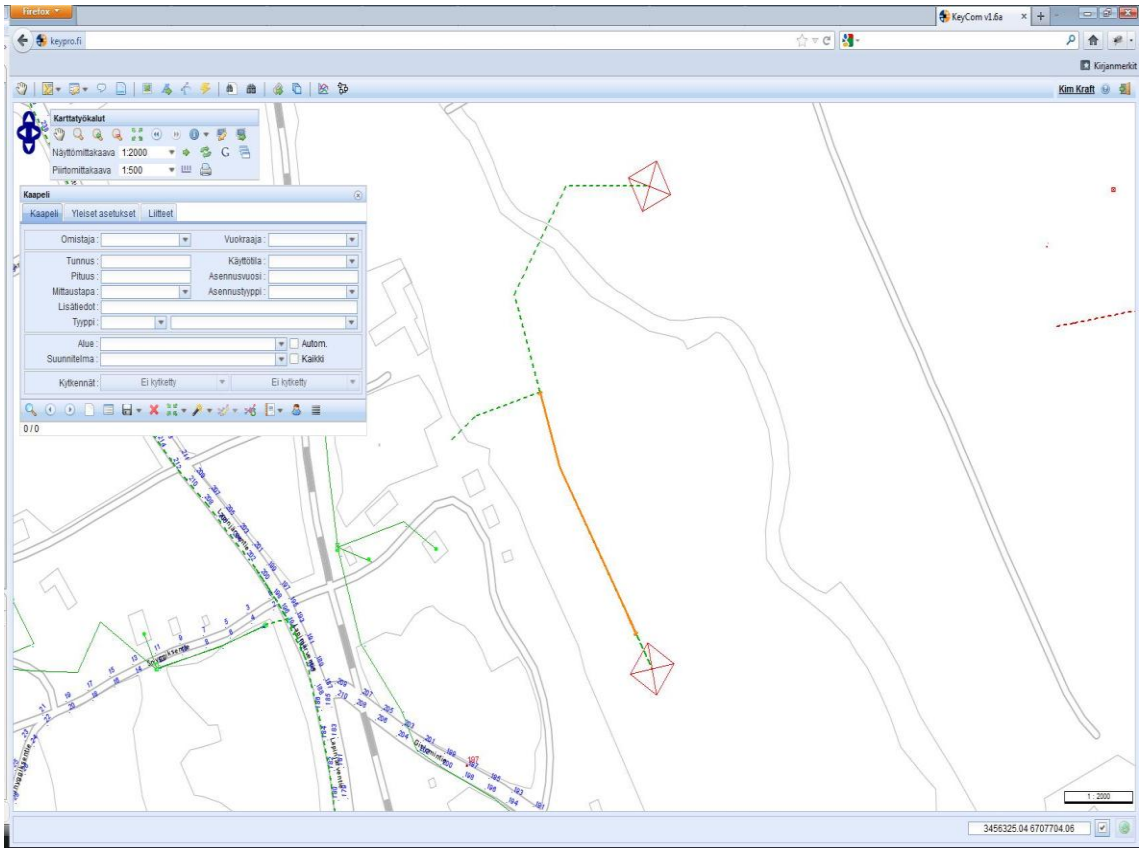
KeyCore itsessään koostuu noin parista kymmenestä erinäisestä Django-sovelluksesta eli osasta, joilla kaikilla on oma tehtävänsä. Keycoren tehtäviin kuuluu muun muassa:

- lomakkeilta tulevien JSON-RPC-pyyntöjen käsittelyt
- verkkotietojärjestelmän alustaminen
- tietokanta kyselyt
- kartan ja karttatasojen luonti, muokkaaminen ja karttaikkunan näyttäminen
- liitännäisten käsittely.

Kaikki KeyCoren päälle rakennetut tuotteet ohjaavat yllä listatut ja monet muut yleiset tehtävät KeyCoren hoidettavaksi. [2, s. 5.]

2.4 KeyCom

Vuonna 2001 Keypro julkaisi teleyrityksille suunnatun KeyRNS-verkkotietojärjestelmän. Kyseisenä vuonna olivat käytössä 56K-modeemit, ja internet oli vasta yleistymässä, joten tuohon aikaan ei voinut puhua nykyajan web-sovelluksista. KeyRNS oli siis työpöytäsovellus, mutta vuonna 2009 Keypro siirtyi web-pohjaiseen ratkaisuun ja julkaisi KeyCom verkkotietojärjestelmän tarkoituksenaan korvata KeyRNS. Nykyään KeyCom on yksi Keypron monista web-pohjaisista verkkotietojärjestelmistä ja se on suunnattu teleyrityksille mahdollistaen muun muassa kuitu- ja kuparikaapeloinnin, kytkentöjen suunnittelun ja dokumentoinnin.



Kuva 1 KeyCom ja kaapelilomake

Kun loppukäyttäjä kirjautuu sisään Keypron verkkotietojärjestelmään, niin pinnan alla selain saa KeyComilta HTML- ja JavaScript-tiedostoja, joiden avulla se luo käyttöliittymän loppukäyttäjän nenän eteen. KeyCom pyytää ensiksi KeyCorelta esimerkiksi kartan, karttatasot, yleiset työkalut ja niin edelleen. Tämän jälkeen KeyCom yksilöi käyttöliittymää muun muassa lisäämällä saatuihin yleistyökaluihin televerkkotyökalut ja palauttaa vastauksen selaimelle, joka luo saatujen tiedostojen avulla käyttöliittymän.

Kuvassa 1 on asiakaspuolen näkymä KeyComista sisäänkirjautumisen jälkeen ja avatusta kaapelilomakkeesta, jonka avulla kartan päälle voi muun muassa piirtää, hakea tai poistaa kaapelia. Keycoren kautta saatiin esimerkiksi karttapohja, kuvan vasemmassa yläkulmassa näkyvät karttatyökalut ja karttaa ympäröivät palkit. Tuote sitten yksilöidään tiettyyn tarkoitukseen suurimmaksi osaksi lomakkeiden avulla. KeyCom-tuotteessa on kaapeli ja ynnä muita telesuunnitteluun liittyviä lomakkeita, kun taas KeyAquasta löytyy esimerkiksi putki- ja pumppaamolomakkeita.

3 Django-sovelluskehys

3.1 Python

Keypro Oy:lla pääsin ensimmäistä kertaa kosketuksiin Python-ohjelmointikielen kanssa. Tämä Monty Python's Flying Circus - televisiosarjasta nimensä saanut kieli on kuuluisa helposta syntaksistaan ja täten suosittu ensimmäinen ohjelmointikieli aloitteleville ohjelmoijille. Itselläni kokemusta ohjelmointikielistä ennen Pythonia oli muun muassa Javasta, C++:sta ja vanhemmasta LISP:stä, joten Python oli muutos kohti selvempää koodia. [16.]

```
6  if tulos == 100:
7      print tulos
```

Koodiesimerkki 1 IF-ehtolause Pythonilla

```
9  if (tulos == 100){
10     System.out.println(tulos);
11 }
```

Koodiesimerkki 2 IF-ehtolause Javalla

Huomattavin ero Pythonin syntaksissa, mikä tekee siitä niin selkeän mielestäni, on sen perustuminen koodin sisentämiseen. Koodissa ei käytetä esimerkiksi aaltosulkeita erottamaan eri lohkoja, vaan esimerkiksi IF-ehtolause ja sen sisältö on erotettu toisistaan sisentämällä, kuten koodiesimerkissä 1 näkyy. Koodiesimerkissä 2 näkyy, että myös Javassa voi sisentää, mutta Pythonin kääntäjä pakottaa sisentämisen toisin kuin Javan, jolloin koodista tulee luettavampaa. Pythonista on myös karsittu sulkeita, rivien lopetusmerkit ja paljon muuta, joka selkeyttää koodia, mutta raju karsiminen verrattuna muiden koodien syntaksiin voi joidenkin mielestä jopa epäselventää koodia.

Olen kuullut, ettei Python ole kaikissa asioissa yhtä kattava ohjelmointikieli kuin esimerkiksi Java tai C++, mutta Pythonia on myös mahdollista laajentaa muilla kielillä, kuten C:llä. [17.]

3.2 Djangon esittely

Django on laaja Python-kirjasto, joka on suunnattu web-pohjaisten sovellusten tekemiseen Pythonilla. Djangon yhtenä perusperiaatteena on tarjota ohjelmoijalle niin sanottuja oikopolkuja, joiden avulla käyttäjän ei tarvitse kirjoittaa samaa koodia useaan kertaan, eikä esimerkiksi huolehtia helposti auki unohtuvista tietokanta yhteyksistä. [5, s. 3-6.]

Arkkitehtuurisesti Django seuraa löyhästi MVC-mallia. Djangon pakottaessa ohjelmoijan seuraamaan Model-View-Controller-mallia. Mallissa sovelluksen kehitys ja toiminnallisuus on jaettu osiin, joissa datan määrittely ja haku on erillään HTTP-pyynnön ja sen ohjauksen logiikasta, joka vuorostaan on erillään asiakaspuolen näkymästä. Sovelluksen jakaminen toiminnallisiin osiin helpottaa sen kehitystä ja antaa selvemmän kokonaiskuvan. Osituksen ansiosta esimerkiksi web-sovelluksen ulkoasun suunnittelijan ei tarvitse ymmärtää Pythonia, vaan hän voi keskittyä sovelluksen HTML- ja CSS-tiedostojen käsittelyyn. [5, s. 3-6.]

Django-sovelluksen perusosat eli Python-tiedostot ja templaatti

- **models.py** – Mallinnustietokannan kentistä Python-luokkana, jossa luokan nimi mallintaa tietokannassa taulun nimeä ja luokkamuuttujat taulun sarakkeita. Yhtä luokkaa kutsutaan ”malliksi” (engl. model) ja mallien kautta suoritetaan kaikki tietokanta operaatiot.

- **views.py** – Näkymä suorittaa jonkin tietyn tehtävän ja palauttaa sen tuloksena templaatin.

- **urls.py** – Tämä tiedosto sisältää URLconf:n joka on ikään kuin sisällysluettelo, joka liittää URL:n ja näkymän toisiinsa.

- **testi.html eli templaatti** – Asiakaspuolen osuus löytyy tästä tiedostosta HTML-tiedostossa. Templaattiin on mahdollista sijoittaa sekaan Djangon omaa templaatti-kieltä, joka mahdollistaa muun muassa IF-ehtolauseiden käytön HTML-kielen seassa. [5, s. 6.]

Keypron verkkotietojärjestelmän palvelinpuoli pohjautuu suurilta osin Djangoon. Satunnaisia PHP- ja PL/SQL-skriptejä löytyy sieltä täältä, mutta Django-palvelimesta voidaan puhua. Tämän opinnäytetyön tekstiviestirajapinta koostuu kaikista yllä listatuista osista templaattia lukuun ottamatta. Templaatin teko eli asiakaspuolen osuus jätettiin rajapintaa käyttävän sovelluksen tehtäväksi. Kappaleessa 5 käydään lyhyesti läpi yksi rajapinnan asiakaspuolen toteutuksista toisen henkilön toimesta, joten templaatit käydään läpi myös tässä kappaleessa.

Seuraavissa kappaleissa käydään yksityiskohtaisemmin läpi normaaliin Django-sovelluksen kuuluvia osia ja on hyvä huomata, että MVC-mallin mukaan Django-sovelluksen ”malli” on **M**odel, ”näkymä” on **V**iew ja ”URLconf” on **C**ontroller. MVC on kuitenkin vain yksi tapa suunnitella web-sovellus ja Django noudattaa sitä löyhästi, joten itse MVC-mallin teoriaan tai historiaan ei pureuduta sen tarkemmin. [5, s. 6.]

3.3 Django-esimerkkisovellus

Tulen käyttämään edellä esimerkkisovellusta kuvaamaan Django-sovelluksen toimivuutta. Oletetaan, että sovelluksen osat sijaitsee polussa */Sovellus/* Django-palvelimen juuressa ja että palvelin toimii osoitteessa *www.esimerkki.fi.*

Djangon mukana tulee testipalvelin, joka on tarkoitettu juuri kehitykseen ja Django-sovellusten testaukseen. Kerron testipalvelimesta hitusen enemmän kappaleessa 6, mutta oletetaan tämän esimerkkisovelluksen kohdalla, että asetukset ovat kohdallaan ja että palvelin toimii täydellisesti. Oletetaan, että tietokanta on olemassa ja yhteys toimii myös täydellisesti.

3.4 Mallit

Django-mallit ovat normaalin Python-luokan näköisiä esityksiä tietokannasta. Koodiesimerkissä 3 on määritelty esimerkkisovelluksen Henkilö-luokka, jonka avulla voidaan tehdä tietokantaoperaatioita kirjoittamatta itse sanaakaan SQL:ää.

```
1  from django.db import models
2
3  class Henkilö:
4      id = models.AutoField(primary_key=True)
5      nimi = models.CharField()
6      puhelin = models.IntegerField()
```

Koodiesimerkki 3 Esimerkki Djangon malli-luokasta

Koodiesimerkin 3 CharField ja IntergerField kertovat Djangolle, minkä tyyppisiä sarakkeita käsitellään tietokannassa. Kun Django-palvelin käynnistetään, niin se tunnistaa uuden mallin eli Henkilö-luokan ja luo automaattisesti uuden Henkilö-taulun tietokantaan, jossa sarakkeina ovat id, puhelin ja nimi.

```
1  p = Henkilö(nimi="Kim", puhelin="050123")
2  p.save()
```

Koodiesimerkki 4 Esimerkki uuden tietueen luomisesta Henkilö-tauluun

Koodiesimerkin 5 ensimmäisellä rivillä luodaan esimerkkinä instanssi Henkilö-luokasta alustaen nimi- ja puhelin-luokkamuuttujat. Django ei vielä tässä vaiheessa ole tehnyt minkäänlaista tietokantaoperaatiota, mutta kun seuraavalla rivillä ajaneen save-funktio, niin Django suorittaa tietokantaoperaation tallentaen juuri luodun instanssin tietokantaan. Id-muuttuja toimii luokan primääriavaimena ja Django automaattisesti kasvattaa siinä olevaa yksilöllistä numeroa uusien tietueitten kohdalla. (Django lisää automaattisesti id-muuttujan malleihinsa, joten sitä ei tarvitsisi erikseen kirjoittaa, mutta se on lisätty koodiesimerkissä Henkilö-luokkaan selvyyden vuoksi.)

Kun save-funktiota kutsutaan, niin taustalla Django hakee palvelimen asetuksista tietokannan tyypin, kirjautumistunnukset, IP-osoitteen ja ottaa yhteyden. Sitten ***INSERT INTO*** *Henkilö (nimi, puhelin)* ***VALUES****('Kim','050123')* SQL-lauseen avulla luo uuden tietueen tietokantaan. [5, s. 86-87.]

Tietokantaoperaatiot ovat yksi hyvä esimerkki aiemmin mainituista oikopoluista, joita Django tarjoaa kehittäjälle. Ohjelmoijan ei tarvitse huolehtia koodissaan tietokanta yhteyksien avaamisesta tai sulkemisesta, eikä puhtaiden SQL-kyselyjen tekemisestä, koska Django tarjoaa aivan oman tietokantarajapinnan, jonka avulla tietokantatoiminnot on automatisoitu ja tehty Pythonin näköisiksi. [5.]

3.5 Templaatit

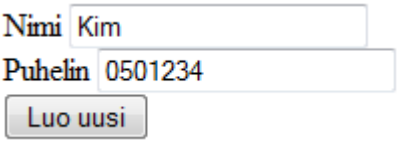
Templaatti on normaali HTML-tiedosto, johon Djangon templaatti-järjestelmä tuo paljon lisämausteita, kuten ehtolausekkeet, silmukkarakenteet ja muuttujat. Djangon kehittäjät ovat pyrkineet modulaarisuuteen, koska sovelluksen visuaalisesta puolesta vastaavan kehittäjän ei pitäisi joutua muokkaamaan Python-koodia tai ylipäätänsä jotain muuta sovelluksen osaa, jolla ei ole mitään tekemistä sovelluksen visuaalisen puolen kanssa. [5, s. 39.]

Edellisessä kappaleessa luotiin malli, joka antaa meille rajapinnan tietokantaoperaatioihin. Seuraavaksi teemme yksinkertaisen luo_henkilo.html-templaatin, jonka sisältönä on HTML-lomake, jonka kautta loppukäyttäjä voi luoda Henkilöitä tietokantaan. Koodiesimerkissä 5 on lomakkeen HTML-koodi ja kuvassa 7 näkyy, miltä lomake näyttää loppukäyttäjän selaimessa.

Lomakkeen kautta loppukäyttäjä voi syöttää nimen ja puhelinnumeron ja kun hän painaa ”Luo uusi”-nappia, kuten kuvassa 2 näkyy, niin lomake lähettää palvelimelle HTTP POST-pyynnön osoitteeseen ”www.esimerkki.fi/luo_uusi/”.

```
1  <html>
2      <head>
3          <title>
4              Luo uusi henkilö
5          </title>
6      <head>
7      <body>
8
9          <form name='luoHenkiloLomake' action = "http://www.esimerkki.fi/luo_henkilo/" method="POST">
10             <label id='nimi'>Nimi</label>
11                 <input type='textfield' id="nimi"> </input></br>
12             <label id='nimi'>Puhelin</label>
13                 <input type='textfield' id="puhelin"> </input></br>
14             <input type='submit' value='Luo uusi'>
15         </form>
16
17     </body>
18
19 </html>
```

Koodiesimerkki 5 luo_henkilo.html-tiedosto henkilön luontiin



Kuva 2 Lomake web-sivulla

Nyt meillä on lomake, jonka kautta loppukäyttäjä luo tietueita Henkilö-tauluun, mutta malli ja templaatti ei itsessään tee vielä mitään. Miten lomake saadaan keskustelemaan Henkilö-mallin kanssa?

3.6 Näkymät

Django view eli näkymä on vastuussa web-sovelluksen sisällön luonnista. Näkymä on palvelinpuolella sijaitseva normaali Python-funktio, jonka avulla tehdään jokin tietty tehtävä, jonka tuloksena palautetaan aina templaatti. Esimerkkisovelluksen kohdalla tarvitsemme näkymän, joka luo uuden tietueen Henkilö-tauluun ja palauttaa vastauksena templaatin, josta käy ilmi, kuinka uuden tietueen luonti onnistui. Tarvitsemme myös näkymän, joka palauttaa vastauksena sovelluksen etusivun. Luomme kaksi näkymää ”index” ja ”luo_uusi”, jotka on nähtävissä koodiesimerkissä 6 . ”index”-näkymään käyttäjä ottaa yhteyden aivan ensiksi, jonka tuloksena käyttäjälle näytetään kuvan 2 mukainen lomake. ”luo-uusi”-näkymän avulla käyttäjä luo uuden tietueen Henkilö-tauluun. [19.]

```
1  from django.http import HttpResponse
2  from django.shortcuts import render
3
4  from sovellus.models import Henkilö
5
6  def index(request):
7      return render(request, 'sovellus/luo_uusi.html')
8
9  def luo_uusi(request):
10     nimi = request.POST['nimi']
11     puhelin = request.POST['puhelin']
12     p = Henkilö(nimi,puhelin)
13     p.save()
14
15     return HttpResponse("Uusi henkilo %s luotu. Puh.: %s" %(nimi,puhelin))
```

Koodiesimerkki 6 Esimerkki sovelluksen näkymät

- Koodiesimerkin 6 ensimmäisellä rivillä otetaan käyttöön HttpResponse-luokka. Djangon vaatimuksena on, että jokaisen näkymän tulee lopuksi palauttaa HttpResponse objekti tai poikkeus (engl. exception). HttpResponse-objekti muunnetaan lopuksi templaatiksi, joten ohjelmoijan ei tarvitse välttämättä tehdä erillisiä templaatti-tiedostoja, vaan Django tarjoaa siihen valmiin pohjan. Rivillä 2 ladataan render-metodi. [19.]

- Rivillä 6 määritellään index-näkymä. Kyseinen näkymä ei tee muuta kuin palauttaa esimerkkisovelluksen luo_uusi.html-templaatin. Tähän näkymään loppukäyttäjä ottaa ensin yhteyttä, jotta hän saisi näkyviin HTML-lomakkeen Henkilöiden luontiin. Render-metodi ottaa kutsussa parametreina: näkymään parametrina annetun request-objektin, palautettavan templaatin ja vaihtoehtoisesti Python-sanakirjan (dictionary). Parametrina saatu request-objekti sisältää HTTP-pyynnön datan muunnettuna Python-objektiksi. Render-metodi ottaa templaatin ja tarkistaa, löytyykö templaatista Djangon templaatti-kieltä, kuten silmukkarakenteita ja lopuksi palauttaa HttpResponse-objektin, jossa on äsken renderöity templaatti sisältönä.

- Rivin 9 ”luo_uusi”-näkymässä tapahtuu varsinainen uuden Henkilön luonti. Tämän näkymän avulla HTML-lomake saadaan keskustelemaan Henkilö-mallin kanssa. Ensiksi otetaan talteen HTTP-pyynnön mukana tulleiden nimi- ja

puhelin-parametrien arvot ja sitten luodaan uusi Henkilö-luokan instanssi ja tallennetaan se tietokantaan, kuten kappaleessa 3.4 käytiin tarkemmin läpi. Lopuksi käyttäjälle palautetaan teksti, että käyttäjä on luotu.

Näin pienessä projektissa luo_uusi.html-templaattia ei tarvittaisi välttämättä ja esimerkeissä käytetyn lomakkeen voisi luoda suoraan index-näkymässä palauttaen luo_uusi.html:n sisältö kokonaisuudessaan HttpResponse objektin mukana, tai käyttämällä Djangon lomake-luokkia. Erillinen templaatti kuitenkin demonstroi, kuinka on hyvä jakaa Django-sovellus osiin. Jos kyseessä on isosta sovelluksesta, niin erillisen templaatin etu tulee paremmin esille. [19.]

Nyt meillä on esimerkkisovelluksen toiminta luotuna, mutta vielä tarvitaan yksi vaihe, jotta Django palvelin osaa ohjata HTTP-pyynnöt oikeisiin näkymiin.

3.7 URLconf

Djangon URLconf on se liima, joka yhdistää käyttäjän pyynnön oikeaan näkymään. URLconf:lla yksinkertaisesti viitataan urls.py-nimiseen tiedostoon, joka sijaitsee palvelimen juuressa tai erikseen palvelimen asetuksissa määritellyssä paikassa. URLconf:a on ikään kuin Djangon sisällysluettelo, jonka avulla Django tietää, mistä sovellus koostuu. Käytännössä URLConf ei ole muuta kuin lista URL:sta ja niitä vastaavista näkymistä. [5, s. 28.]

```
1  from sovellus.views import index, luo_uusi
2
3  from django.conf.urls.defaults import *
4
5  urlpatterns = patterns("",
6                         '(^$)', index),
7                         '(^luo_uusi\$)', luo_uusi))
```

Koodiesimerkki 7 Esimerkki sovelluksen urls.py tiedoston sisältö

Jatkamme edellisistä kappaleista tutulla esimerkkisovelluksella. Koodiesimerkissä 7 näkyy sovelluksen urls.py tiedosto eli sen URLconf. Kun Django-palvelimelle saapuu

HTTP-pyyntö, niin se vertailee saapuneen pyynnön URL:a URLconf:sta löytyvään urlpatterns-muutujan URL:hin. Jos vastaavanlaisuus löytyy, niin Django kutsuu URL:a vastaavaa näkymää antaen argumenttina HttpRequest-objektin eli saapuneen HTTP-pyynnön. Tästä johtuen näkymiä kirjottaessa pitää aina muistaa ottaa vastaan HttpRequest-objekti parametrina, kuten koodiesimerkissä 6 näkyy. [5, s.28.]

URL-vertailu tapahtuu säännöllisten lausekkeiden (engl. regular expression) perusteella. Koodiesimerkissä 7 rivillä 6 säännöllinen lauseke ”^$” vastaa tyhjää riviä eli palvelimen juurta. Kun loppukäyttäjä suuntaa esimerkki sovelluksen URL:n ”www.esimerkki.fi”, niin pyyntö ohjataan index-näkymään, loppukäyttäjä saa eteensä aikaisemmissa kappaleissa luodun lomakkeen, joka myös näkyy kuvassa 2. Koodiesimerkissä 5 näkyy, että asetimme lomakkeen action-attribuuttiin arvon ”www.esimerkki.fi/luo_uusi/” eli kun käyttäjä painaa lomakkeelta löytyvää nappia, niin HTTP-pyyntö lähtee kohti kyseistä osoitetta, jossa Django ohjaa pyynnön luo_uusi-näkymään. Jos loppukäyttäjä yrittää yhdistää osoitteeseen, jota ei ole määriteltyä URLconf:ssa, niin Django tuottaa oletuksena 404-tyyppisen HTTP-virheen. [5. s.26-28.]

Näin valmistui yksinkertainen Django-sovellus ja tarkoituksena oli antaa lukijalle suuntaa, kuinka Django toimii. Raapaisimme vain pintaa Djangosta, mutta templaatteja lukuun ottamatta kaikkia kappaleessa 3 käsiteltyjä osia käytettiin myös tämän insinöörityön tekstiviestirajapinnan luonnissa.

3.8 Järjestelmävalvojan sivu

On hyvä vielä mainita Djangon mukana tulevasta järjestelmävalvoja-sivusta (engl. admin-page), joka on graafinen web-käyttöliittymä Django-palvelimen ja sen sovellusten hallintaan. Käyttöliittymä kautta voi muun muassa tarkastella sovellusten tietokantojen sisältöjä, luoda uusia järjestelmävalvojia ja ryhmiä. Jokaiselle sovellukselle täytyy ottaa järjestelmävalvojan sivu erikseen käyttöön, jotta tietokantaoperaatioita voisi suorittaa sovelluskohtaisesti.

```
1  from django.contrib import admin
2  from sovellus.models import Henkilö
3
4  admin.site.register(Henkilö)
```

Koodiesimerkki 8 Esimerkki sovelluksen admins.py tiedosto

Oletetaan, että aiemmin esimerkkinä olleella palvelimella on asetuksista määritelty käyttöön järjestelmävalvoja-sivu. Jos haluamme, että esimerkkisovellus näkyy myös kyseisellä sivulla, niin joudumme luomaan admins.py-tiedoston sovelluksen juureen. Sovelluksen admins.py-tiedoston sisällön voi nähdä koodiesimerkistä 8 ja kuten rivillä 2 näkyy, niin itse asiassa lataamme vain sovelluksen Henkilö-mallin ja rekisteröimme sen palvelimen järjestelmävalvojasivulle rivillä 4. Pienellä lisävaivalla uusia Henkilöitä voi nyt luoda käyttäen graafista käyttöliittymää. [5, s.102.]

4 Tekstiviestirajapinnan suunnittelu

4.1 Tekstiviestirajapinnan määrittely

Aluksi minulle annettiin projektiksi luoda tekstiviestirajapinta KeyCore:n ilman sen tarkempia määrittelyjä ja aloin heti tuumasta toimeen. Minulla oli kuitenkin paljon opettelemista perusohjelmointityökaluista lähtien, joten sain ensimmäisinä viikkoina hyvin vähän aikaan itse projektin osalta. Tämän aikana esimieheni Joensuussa antoi minulle tarkemmat määrittelyt. Hän osoitti minut oikeaan suuntaan. [18.]

Yleinen kuvaus tekstiviestirajapinnan toiminnasta oli, että *rajapinta saa pyynnön, jossa pyydetään lähettää tekstiviesti pyynnön mukana tulleilla parametreilla, ohjelma tallentaa alustavan pyynnön tietokantaan, lähettää viestin internetin yli SMS-yhdyskäytävään ja päivittää tietokantaan mikä viestin lähetys tilanne on.* [18.]

Viikon työskentelyn jälkeen lähetin ensimmäisen version tekstiviestirajapinnan lähdekoodista Joensuuhun, jossa esimieheni luki koodini läpi ja antoi todella rakentavaa palautetta. Määrittelyjä parannettiin tämän myötä ja koko rajapinta jaettiin eri luokkiin, joilla oli eri rooli rajapinnan toiminnassa. [18.]

Tekstiviestirajapinnan eri osat eli luokat

- **MessagingService** – Rajapinnan ydin. Tämän luokan kautta suoritetaan kaikki tekstiviestioperaatiot.
- **Adapter (abstrakti-luokka)** – Jokaiselle eri SMS-yhdyskäytävän palveluntarjoajalle tehdään aina yksilöllinen Adapter-luokka. Näitä luokkia voi yleisesti nimittää Adapter-luokiksi, mutta ne ovat aina toiminnaltaan ja nimeltään operaattori kohtaisia.
- **AdapterReponse** – Kaikki Adapter-luokat muodostavat vastauksen tämän luokan luokkamuuttujien perusteella
- **MessageRequestLog** – Djangon malli-luokka. Tietokanta operaatiot tämän kautta
- **SmsParameters** – Tekstiviestin lähetystiedot luodaan tästä luokasta luotavan instanssin avulla.

[18.]

Näiden määrittelyjen avulla rajapinnan rakenne selkeni huomattavasti, ja itselläkin oli nyt selvä suunta ohjelmoinnin puolesta. Ennen varsinaista ohjelmointia täytyi suunnitella rajapintaa vielä tarkemmin yllä olevien määrittelyjen pohjalta.

4.2 Modulaarisuus ja geneerisyys

Kun minulle ensimmäisinä annettiin projektiksi tehdä tekstiviestirajapinta, niin aloin ohjelmoida rajapintaa sen enempää sitä suunnittelematta. Suunnittelun tärkeyttä on painotettu jo koulun ensimmäisestä ohjelmointikurssista lähtien, mutta kärsivällisyyteni ei riittänyt suunnittelemaan kunnolla, jolloin esimerkiksi ohjelman koodausvaiheet eivät olleet johdonmukaisia. Koska ohjelmoin eri osia sieltä sun täältä, joku olennainen osa jäi helposti tekemättä tai keskeneräiseksi, mikä hidasti projektin etenemistä. [18.]

Rajapinnan suunnittelussa piti ottaa huomioon kaksi ohjelmointimaailman sivistys-sanaa ”modulaarisuus” ja ”geneerisyys”. Esimieheni vilkaistua koodiani ensimmäisen kerran, hän halusi, että koodia pitää ositella vielä enemmän eli luoda luokkia toiminnallisuuden mukaan ja luokissa tulee käyttää funktioita, joilla on myös jokin selvä tehtävä, jota funktion nimi selvästi kuvaa. Funktiot eivät mielellään saisi olla yli 20 riviä

pitkiä, eivätkä eri luokat saa olla liian riippuvaisia toisistaan eli muutos tiettyyn luokkaan ei saisi vaikuttaa muiden luokkien toimivuuteen millään tavalla tai hyvin vähän. Tämä modulaarisuus eli rajapinnan paloitteleminen pieniin toiminnallisiin osiin tekee siitä helposti muokattavan tulevaisuudessa ja helpottaa hahmottamaan rajapinnan toimintaa. Modulaarisuuden lisäksi rajapinnan piti olla geneerinen eli ihan mikä tahansa Django-sovelluksen pitäisi pystyä käyttämään rajapintaa vaivattomasti ja ihan sama mitä kautta tekstiviestit loppujen lopuksi lähetetään, niin rajapintaa käytetään aina samalla tavalla. Myöhemmin selviää, että geneerisyys saavutetaan modulaarisuuden kautta tämän rajapinnan kohdalla. [18; 22.]

Aikaisemmin tehdyn tekstiviestirajapinnan määrittelyn perusteella suunnittelimme yhdessä esimieheni kanssa luokkarakenteen, joka pätkii rajapinnan osiin. Taulukossa 1 on rajapinnan luokkarakenne, josta käyvät ilmi luokat, luokkamuuttujat ja luokkien funktiot eli metodit. Kappaleessa 5 käydään läpi, mitkä luokat keskustelevat keskenään ja mihin eri metodeja käytetään. [6.]

Haluan näin alussa tuoda esille, että koko suunnitelma ja toteutus tehtiin yhteistyössä esimieheni kanssa. Tässä luvussa käsiteltävä lopullinen suunnitelma muodostui useiden keskustelujen myötä ja muokkaantui myös itse toteutuksen aikana. Luvusta voi oppia, minkälainen ihanteellinen suunnitelma olisi ollut rajapinnan toteuttamiseen. Seuraavissa luvuissa käydään läpi luokkakohtaiset suunnitelmat.

Taulukko 1 Tekstiviestirajapinnan luokkarakenne

Luokka	Luokan funktiot	Luokkamuuttujat
MessagingService (Rajapintaa käytetään tämän luokan kautta.)	__init__ startup send_sms update_sms_status	backend (String) started (Boolean)
Adapter (Adapter-luokka on operaattorikohtainen ja sen toiminta logiikka muuttuu operaattorin vaatimusten mukaan.)	__init__ startup simple_send_sms parse_http_response	username (String) password (String) source_name (String) URL (String)

MessageRequestLog (Tietokanta-luokka. Luokkamuuttujat mallintavat suoraan tietokannan kenttiä)		request_id request_created request_updated request_by request_status message_recipients text_message message_sender request_response message_type message_category message_origin messages_accepted
AdapterResponse (Adapter-luokat muodostavat vastauksena tämän luokan luokkamuuttujien kautta.)		SERVICE_RESPONSE = "" STATUS_OK = 0 STATUS_ERROR = 1 RECIPIENT_OK = 0 RECIPIENT_ERROR = 1 RECIPIENT_ERROR_INVALID = 2 RECIPIENT_ERROR_DUPLICATE = 3 RECIPIENT_ERROR_UNALLOWED = 4 RECIPIENT_ERROR_ROUTING_ERROR=5
SmsParameters (Tekstiviestien lähetys tapahtuu tämän luokan luokkamuuttujien avulla. MessagingService osaa lähettää tekstiviestin vain tämän luokan luokkamuuttujien avulla.)		sender = None request_by = None message_origin = None recipients = [.] sms_type = u'normal' message = None

4.3 MessageRequestLog - Luokka tietokantaoperaatioihin

Suunnitelmaan kuului, että lähetetyt tekstiviestit tallennettaisiin tietokantaan laskutus tarkoitukseen, ylläpitoon ja valvontaan. Kuten kappaleessa 3 käsiteltiin, niin Django tarjoaa erinomaisen rajapinnan tietokantakyselyihin, joten ohjelmointiosuus tietokannan osalta ei pitäisi tuottaa hirveästi vaivaa. Tietokanta taulua mallintavan luokan nimen tuli olla tarpeeksi sisältöään kuvaava ja englanniksi, joten päädyimme MessageRequestLog nimeen.

Tarkoituksena oli, että kun rajapintaan tulee tekstiviestin lähetyspyyntö tietyillä parametreilla, niin lähetyspyynnön tiedot tallennettaisiin alustavasti tietokantaan ennen varsinaista tekstiviestin lähetystä, ja tätä tallennettua tietoa päivitettäisiin tekstiviestin lähetyksen jälkeen. Vaikka Djangon avulla tietokantaoperaatiot ovat helppoja, niin piti miettiä, mitä tietoa tekstiviestin lähetyksestä on hyvä tallentaa?

MessageRequestLog-luokalla tuli olla tarpeeksi eri luokkamuuttujia, jotta tietoa olisi tarpeeksi laskutusta, ylläpitoa ja valvontaa varten. Kuten luvussa 3.4 mainittiin, niin luokkamuuttujat Djangon tietokantaluokassa mallintavat itse asiassa sarakkeita tietokantataulussa. Hyvä tapa luokkamuuttujien keksimisessä oli esittää kysymyksiä, jotka voisivat tulla asiakkaan tai ylläpidon mieleen halutessaan tietoa tekstiviestistä tai sen lähetyksen tilasta, ja kysymysten perusteella luoda luokkamuuttujia, joista löytyy vastaus.

- Milloin viestipyyntö on saapunut ja milloin pyyntöä on päivitetty?
- Kuka lähetti viestin ja kenelle?
- Mikä on viestin lähetyksen tilanne?
- Mitä kirjoitin tekstiviestiini kuukausi sitten?
- Montako viestiä olen lähettänyt?
- Onko lähettämäni viestit saapuneet perille, jos ei niin miksi?

Kysymysten ja yleisen pähkäilyn tuloksena luotiin yhteensä 13 luokkamuuttujaa, jotka käyvät ilmi taulukosta 1.

4.4 MessagingService - Tekstiviestirajapinnan julkinen osa

MessagingService on tekstiviestirajapinnan julkinen luokka, jonka kautta ulkopuoliset sovellukset käyttävät rajapintaa. Ainut julkinen metodi, joka tarjotaan on send_sms-metodi, jonka avulla lähetetään tekstiviesti. MessagingService-luokasta haluttiin geneerinen, eli rajapinnan käyttäjä lähettää luokan kautta tekstiviestejä, mutta käyttäjän ei tarvitse huolehtia mitä kautta tai miten tekstiviestit lähetetään. Käyttäjän tulee vain antaa send_sms-metodin kutsussa argumenttina tekstiviestin lähetykseen tarvittavat perustiedot ja ottaa vastaan vastaus lähetyksen tilasta. Nämä niin sanotut perustiedot annetaan send_sms-metodin kutsussa argumenttina, jonka tulee olla SmsParameters-luokan instanssi.

SmsParameters-luokassa määritellään, mitkä tiedot vaaditaan, jotta send_sms-metodi osaa lähettää viestit eteenpäin. Yleensä kun metodi vaatii vain muutaman argumentin, niin ne voi määritellä suoraan __doc__-osiossa, mutta kun vaadittujen argumenttien määrä kasvaa suureksi, niin on kätevää määritellä erillinen luokka ja mainita __doc__ osiossa, että ”anna argumenttina SmsParameters-luokan instanssi”. Itse argumentti-luokassa voi sitten tarkemmin kuvailla luokan ja luokkamuuttujien tarkoitusta. Tämä helpottaa tulevaisuudessa vaadittujen argumenttien lisäämistä tai poistamista ja pitää asiat selvempinä.

Send_sms-metodin tuli muun muassa trimmata vastaanottajien puhelinnumerot eli karsia numeroista ’-’ merkit pois ja kaikki ylimääräiset välit. Metodin täytyi myös hoitaa tekstiviestien tallennus tietokantaan käyttäen MessageRequestLog-luokkaa ja palauttaa vastaus viestin lähetyksen onnistumisesta tai epäonnistumisesta.

MessagingService-luokka ei kuitenkaan ollut se viimeinen vaihe tekstiviestin lähetyksessä, vaan rajapinnan tulee pystyä keskustelemaan eri operaattorien kanssa, joten rajapinta tarvitsisi lisäosan jokaista operaattoria kohden. Jokaista operaattoria kohden luotaisiin Adapter-luokka, jonka avulla MessagingService-luokka lähettäisi tekstiviestejä tietyn operaattorin kautta.

4.5 Adapter-luokat

Keyprolla oli harkinnassa, että varsinainen tekstiviestin lähetys matkapuhelinverkkoon hoidettaisiin heidän omissa tiloissaan, mutta ajan puutteen vuoksi päädyttiin käyttämään ulkoisia SMS-palveluntarjoajia, jotka tarjoavat internetissä niin sanotun SMS-oletusyhdyskäytävän (gateway), jonka kautta voi lähettää tekstiviestejä.

Adapter-luokat erikoistuvat tekstiviestipyynnön lähetykseen internetin yli tietylle SMS palveluntarjoajalle. Tätä suunnitelmaa tehtäessä Keyprolla oli sopimus kahdelle SMS palveluntarjoajalle, Labyrintti Media Oy:lle ja Tietokoura Oy:lle, joten suunnitelmaan kuului luoda kaksi Adapter-luokkaa, joille päätettiin antaa nimeksi LabyrinttiAdapter ja TietokouraAdapter.

Adapter-luokan tehtävänä on ottaa MessagingService-luokalta tuleva tekstiviestipyyntö ja valmistella pyyntö lähetettäväksi SMS-palveluntarjoajalle. Vaadittu HTTP-pyynnön sisältö poikkeaa täysin toisistaan Labyrintin ja Tietokouran välillä. Kyseiset palveluntarjoajat vastaanottavat täysin eri parametreja ja käyttävät eri tekniikkaa vastaanottamaan niitä. LabyrinttiAdapter-luokan täytyi lähettää tekstiviestipyynnön parametrit käyttäen HTTP POST-metodia, kun taas TietokouraAdapter-luokan tuli käyttää SOAP-kutsuja, joten jo tässä vaiheessa operaattorikohtaisten Adapter-luokkien käyttö osoitti kyntensä, koska näiden toteuttaminen suoraan MessagingService-luokkaan tekisi siitä sekä sekavan ja ei geneerisen luokan.

Kun adapter luokat ovat irrallisia MessagingService-luokasta, niin tulevaisuudessa on paljon helpompi luoda uusia Adapter-luokkia mahdollisille uusille SMS-palveluntarjoajille tai jos Keypro siirtyy käyttämään omaa SMS-palvelinta, niin siirtymä siihenkin on vaivattomampi. Adapter-luokkien tulee kuitenkin aina palauttaa MessagingServicen ymmärtämä vastaus, koska jos jokainen Adapter-luokka palauttaisi erilaisia vastauksia, niin uutta Adapter-luokkaa tehtäessä joutuisimme muokkaamaan myös MessagingService-luokkaa, jolloin tämä sotisi rajapinnan modulaarisuutta vastaan.

Taulukon 1 AdapterResponse-luokka luotiin nimensä mukaisesti Adapter luokkien vastausluokaksi ja oli tarkoitukseltaan samanlainen kuin aiemmin mainittu SmsParameters-luokka.

5 Tekstiviestirajapinnan toteutus

5.1 Työkalut

Ensimmäiset päivät menivät ihan työkalujen asentamisessa ja Keypron ympäristön opettelemisessa. Seuraavana on listaus työkaluista, joita käytettiin tekstiviestirajapinnan kehittämiseen.

Eclipse

Eclipse on varmasti kaikille vähäkin ohjelmointia harrastaneelle tuttu kehitystyökalu. Itselläkin oli aiempaa kokemusta Eclipsestä Java ohjelmoinnin kautta, mutta Eclipse taipuu lukuisille ohjelmointikielille, kuten Pythonille. Eclipse ei kuitenkaan oletuksena tue Pythonia, joten Eclipsen tuli asentaa PyDev lisännäinen, joka mahdollisti Python kehityksen tukien myös Djangoa. [7; 8.]

Mercurial

En ollut ennen käyttänyt minkäänlaista versionhallintajärjestelmää, se oli käsitteenäkin vähän outo. Ymmärsin kuitenkin nopeasti, että jonkinlainen versionhallinta järjestelmä on pakko olla olemassa ohjelmointitalossa, jotta lukuisat ohjelmoijat voivat työskennellä saman projektin parissa vaivattomasti.

Versionhallintajärjestelmässä kehitettävän sovelluksen tuotannossa käytettävä lähdekoodi on ladattuna yrityksen palvelimelle, josta jokainen ohjelmoija lataa kopion omalle työasemalle. Ohjelmoija työskentelee paikallisen kopionsa parissa ja kun hän on saanut työnsä päätökseen, niin hän kopioi palvelimelta uusimman version lähdekoodista, ”sulattaa” uusimman version lähdekoodista omaan muokattuun lähdekoodiinsa korjaten mahdolliset ristiriidat ja siirtää tämän sulatetun version takaisin palvelimelle. Testauksen jälkeen tämä sulatettu versio sulatetaan osaksi tuotannossa käytettävää sovellusta. Yksinkertaisen nerokasta. [10.]

Mercurial on vain yksi monista versionhallintajärjestelmistä, mutta näin versionhallintajärjestelmien noviisille Mercurial oli varsin helppokäyttöinen ja erittäin kätevä. Mercurialin avulla omat virheeni eivät vaikuta muitten työskentelyyn, pystyn

tallentamaan työni palvelimelle ja jos jotain meni pahasti pieleen, täten rikkoen sovelluksen, niin pystyin palauttamaan paikallisen kopioni siihen tilaan, jossa se vielä toimi. [9.]

Python 2.7

Normaalin Python 2.7-asennuksen lisäksi käytössä oli lukuisia Python-lisäkirjastoja, kuten muun muassa PIL, GDAL ja tietenkin Django.

iPython

Paranneltu komentorivi Python-koodin ajamiseen. iPythonilla on kätevä testata pieniä osia sovelluksesta, kuten esimerkiksi uuden funktion testaus, jotta se toimii odotetulla tavalla.

Redmine

Keypron projektien hallinta perustuu tikettijärjestelmään. Jokaisesta tehtävästä, jonka parissa joku ohjelmoijista työskentelee, täytyy olla tiketti olemassa. Tikettiin ohjelmoija kirjaa tehtävän vaiheita, lisää liitteitä ja dokumentoi kaikki taivaan ja maan väliltä.

Redmine on web-pohjainen projektin hallintatyökalu, josta löytyy kyseinen tikettijärjestelmä ja monia muitakin osia. Redminessä on muun muassa Keypron Wiki-sivut, mistä löytyvät ohjeet esimerkiksi yllä olevien työkalujen asentamiseen.

Djangon testiympäristö

Djangon testiympäristön avulla ohjelmoija varmistaa lähdekoodin toimivuuden kokonaisuudessaan ja suojelee koodia myös itseltään. Testiympäristöstä kerrotaan enemmän luvussa 6.

5.2 Hyvät tavat ohjelmoinnissa

Hyviä ohjelmointitapoja on useita, joista jotkut ovat ihan makuasioita, mutta käsittelen seuraavaksi muutaman tavan, joita noudatin tai aloin noudattaa ohjelmoinnin edetessä.

Ohjelmoitaessa tuli muistaa kirjoittaa koodiin loggereita, jotka kertoivat ohjelman etenemisestä Eclipsen konsolissa. Näin tiedetään, suoriutuiko ohjelma loppuun asti ja jos ei, niin loggerien avulla pystyy suunnilleen päättelemään, missä kohtaa ohjelma kaatui. Loggereiden avulla saadaan myös tietää, mitä palvelimen sisällä oikein tapahtuu, koska muuten palvelimen ei hirveästi avaudu omista asioistaan.

Toinen tärkeä asia oli muistaa käyttää hyväksi Django-testaustyökaluja. Aina kun kirjoitin uuden osan rajapintaan, niin kirjoitin osalle myös testin, joka testasi että homma toimii niin kuin pitää. Jos ohjelman jokin osa hajoaa, niin se on helppo jäljittää ajamalla testi, joka kertoo missä kohtaa ohjelma lakkasi toimimasta. Myös joku muu kuin alkuperäinen ohjelmoija pystyy jäljittämään vian paljon nopeammin kunnon testin avulla. Aiemmin mainitut loggeritkin antavat myös suuntaa ohjelman jäätymiskohtiin, mutta testeistä käy ilmi, miten alkuperäinen ohjelmoija halusi ohjelman toimivan.

Kolmantena hyvänä tapana oli muistaa kommentoida koodia. Itse pyrin noudattamaan tätä tapaa, mutta huomasin, että mitä enemmän opin ohjelmoimaan Pythonilla, niin sitä vähemmän kommentoin koodia ja lukiessani muiden kokeneiden ohjelmoijien koodia, niin kommentointi oli lähes olematonta. Ehkä syy tähän on, että ohjelmoija kommentoi vain koodia, joka on vähän epäselvää hänelle, mutta ohjelmoijan kehittyessä tietenkin myös epäselvien osuuksien lukumäärä vähenee. On kuitenkin hyvä vähintään kommentoida, mitä eri osuuksien on tarkoitus tehdä koodissa. Jos esimerkiksi koodissa on virhe, eikä se toimi niin kuin kommentissa on sanottu, niin toinenkin ohjelmoija voi helpommin korjata koodin ja jopa parantaa sitä, jos tiedetään mitä alkuperäinen ohjelmoija on tavoitellut.

Python-kieli on selvä luettavuuden osalta, koska se pakottaa ohjelmoijan sisentämään koodia. Python-kappaleessa mainittiin, että syntaksi perustuu sisentämiseen, joten Python-ohjelma ei yksinkertaisesti toimi, jos sitä ei ole sisennetty oikein. Muut ohjelmointikielet, kuten Java tai C++ eivät tätä vaadi, ja kokematon ohjelmoija saa koodin halutessaan todella vaikealukuiseksi. Python itse opettaa ohjelmoijan hyville tavoille tässä suhteessa.

Seuraavissa kappaleissa tullaan näyttämään kuvia lähdekoodista, jossa kommentti-rivit alkavat #-merkillä ja yllä mainitut loggerit noudattavat muotoa ”logger.debug(’ Viesti ohjelman etenemisestä’)”.

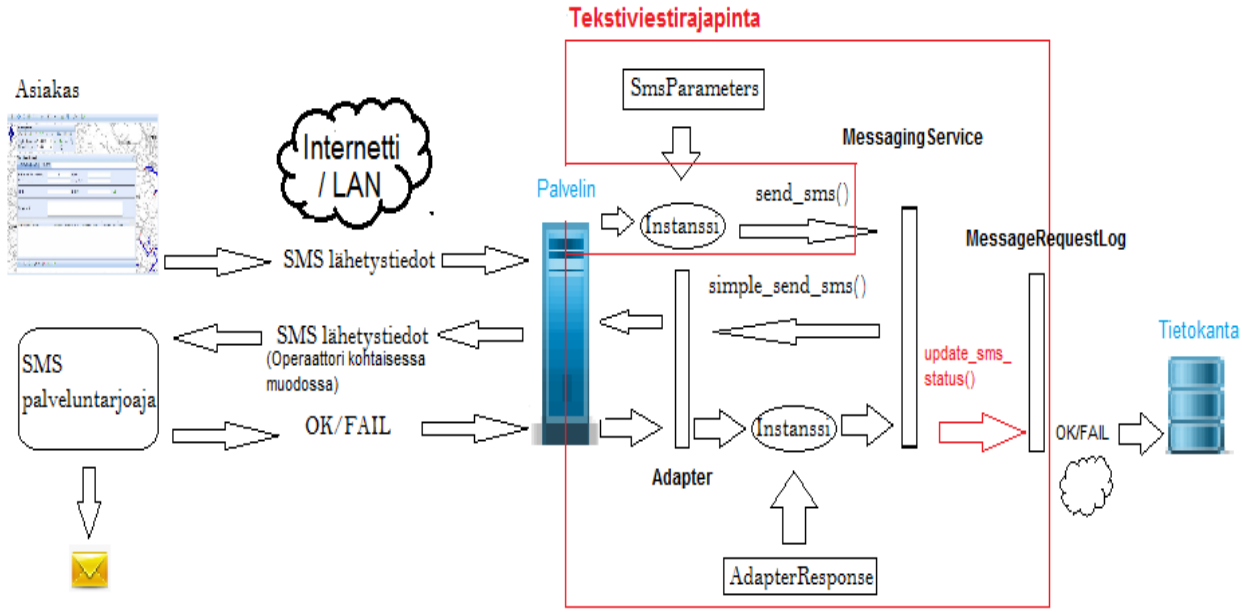
5.3 Toteutus

Seuraavaksi käydään läpi, minkälainen tuotos saatiin aikaan suunnitelman perusteella. Kuvasta 6 käy ilmi, kuinka tekstiviestin lähetys etenee kokonaisuudessaan asiakaspuolelta eli selaimesta aina SMS-palveluntarjoajalle asti. Vaikka tämä toteutus-luku keskittyy varsinaisen tekstiviestirajapinnan eri osiin, niin on hyvä saada kuvan mukainen kokonaiskuva eri vaiheista ja siitä, missä välissä itse tekstiviestirajapinta vaikuttaa.

Käydään ensiksi yleisesti läpi kuvan 6 vaiheita. Kuvassa nuolet näyttävät pyynnön etenemis-suunnan, joista punaisella rajatun alueen eli tekstiviestirajapinnan ulkopuolella kuvaavat internetin tai lähiverkon yli tapahtuvaa siirtoa ja vastaavasti rajatun alueen sisäpuolella kuvataan varsinaisen tekstiviestirajapinnan eri vaiheita. Nuolien päällä olevat tekstit kertovat, millä metodilla ohjelman suoritus siirtyy luokasta toiseen, ja nuolien välissä tai vieressä on kuvaus kuljetettavasta datasta. Esimerkiksi ”SMS lähetystiedot” kuljettaa tekstiviestin lähetykseen tarvittavia tietoja HTTP-paketissa. update_sms_status-metodi on poikkeus ja täten punaisella, koska kyseinen metodi ei siirrä ohjelman suoritusta varsinaisesti toiseen luokkaan, vaan se suorittaa tietokantaoperaation MessagingService-luokan sisällä, mutta käyttäen MessageRequestLog-luokkaa. On myös hyvä huomata rajapinnan viivan ylittävät nuolet, jotka kuvaavat rajapinnan ja sitä käyttävän ulkopuolisen sovelluksen välistä keskustelua. Pystyssä ja vaakasuorassa olevat suorakulmiot ovat rajapinnan luokkia, joiden nimet on kerrottu joko suorakulmion sisällä tai vieressä, ja ovaalit ovat luokista luotuja instansseja. Kuvassa näkyy myös, kuinka SMS-palveluntarjoaja vastaa palvelimelle takaisin kertoen, onnistuiko tekstiviestin lähetys vai ei ja tämä vastaus tallennetaan tietokantaan. [6.]

Kun SMS-lähetystietoja sisältävä HTTP-pyyntö saapuu palvelimelle asiakaspuolelta, niin palvelin osaa ohjata ohjelman suorituksen pyynnössä olevan URL:n avulla oikeaan Django-näkymään, joka luo pyynnön mukana tulleiden lähetystietojen perusteella SmsParameters-instanssin ja kutsuu MessagingService-luokan sends_sms-metodia

antaen argumenttina luodun instanssin, jonka jälkeen on rajapinnan vuoro loistaa.



Kuva 3 Tekstiviestin lähetyksen vaiheet rajapinnan kautta

5.3.1 Rajapinnan alustus

Ennen kuin kuvassa 6 näkyvä send_sms-metodi tai ylipäätänsä koko rajapinta on käytettävissä, niin täytyy rajapintaa käyttävän näkymän luoda, koodiesimerkin 9 mukaisesti, instanssi MessagingService-luokasta ja ”käynnistää” rajapinta.

```
154     #initialization
155     service = MessagingService()
156     service.startup()
```

Koodiesimerkki 9 Rajapinnan käyttöönotto Django-näkymässä

```
class MessagingService(object):
    """ Class for sending SMSs
        Sending SMS: 1) Call startup() (at least once) to make sure service is up and running 2) Define parameters in SmsParameters class
    """

    def __init__(self, backend=None):
        """
            Note: constructor should not do anything "heavy", no accessing database or initializing backends.
            Use initialize method to perform real initialization as it might be potentially a very "heavy" operation.
            Arguments:
                backend - optional, string, default None, fully qualified backend name
                    Example: 'keycore_messaging.adapters.tietokoura.SmsService'
                    System will use backend defined by the KEYCORE_MESSAGING_BACKEND if this argument is omitted or None
        """

        self.backend = None
        self.backend_name = backend or settings.KEYCORE_MESSAGING_BACKEND
        logger.debug("Messaging service will use '%s' backend", self.backend_name)
        #Load and create instance of the backend class,
        # this must be fast and safe operation, as adapter construction should not do much
        self.backend = resolve_class_from_str(self.backend_name)()
        self.started = False
```

Koodiesimerkki 10 MessagingService-luokan __init__-alustusmetodi.

Instanssin luonti rajapinnasta tekee sille alustustoimintoja perustuen palvelimen tai käyttäjän määrityksiin. Koodiesimerkissä 10 näkyy, että MessagingService-luokalle, kuten kaikille Python-luokille on mahdollista kirjoittaa alustus-metodin nimeltään __init__, jolla tehdään alustavia toimintoja luokalle aina kun siitä luodaan instanssi. Alustus-metodi ei ole pakollinen, mutta Python-tulkki kutsuu sitä automaattisesti, jos sellainen löytyy. On myös huomata koodiesimerkissä esiintyvä self-parametri. Kaikki Python-metodit saavat ensimmäisenä parametrina viittauksen luokkaan, tai jos metodi ei ole luokan sisällä, niin viittaus on Python-moduuliin, jossa metodi sijaitsee. Metodin ensimmäiselle parametrille on tapana antaa nimeksi ”self”. Self-parametrin avulla voi esimerkiksi alustaa luokkamuuttujia tai kutsua luokan/moduulin muita metodeja metodin sisältä. Haluan painottaa, että kyseessä on enemmänkin kirjoittamaton sääntö, että nimeksi annetaan ”self”, vaikkei Python tulkki pakota sitä millään tavalla.[23.]

Koodiesimerkissä 10 olevan rajapinnan __init__-metodissa alustetaan backend-luokkamuuttuja käyttäen, joko parametrina annettua backend-muuttujaa tai palvelimen asetuksista löytyvää määrittelyä. Backend muuttuja tulee sisältämään alustuksen

jälkeen Adapter-luokan instanssin. Myös started-luokkamuuttuja alustetaan Falseksi ja sitä käytetään merkkaamaan, onko rajapinnan startup-metodia kutsuttu aiemmin. [23.]

Jokaisella Django-palvelimella on settings-tiedosto, jossa määritellään kaikki Django-kohtaiset asetukset. Settings-tiedosto on itse asiassa normaali Python moduuli, johon Django-sovellukset voi viitata itse koodissa Djangoon sisäänrakennetun settings-luokan avulla. Esimerkiksi koodiesimerkissä 10 rivillä 95 asetetaan backend_name-muuttujalle arvoksi, joko metodille parametrina annettu backend-muuttujan arvo, tai vaihtoehtoisesti asetuksista löytyvä KEYCORE_MESSAGING_BACKEND-muuttujan arvo. Backend_name-muuttujan tulee saada arvoksi merkkijono polusta, mistä haluttu Adapter-luokka löytyy ja rivillä 99 backend_name-muuttujasta löytyvän polun avulla luodaan instanssi Adapter-luokasta. [11.]

```
102     def startup(self):
103         """
104             Starts up the SMS service by initializing the backend.
105             May be slow.
106             May crash if various errors.
107             Normally should be done only once.
108             Will have self.started set to True on successful startup (can be checked if this instance has been started already)
109         """
110         #Initialize backend, potentially slow operation
111         self.backend.startup()
112         self.started = True
113         logger.debug("Messaging service has been successfully started")
```

Koodiesimerkki 11 MessagingService-luokan startup-metodi

```
29      def startup(self):
30          """
31              Prepares the adapter for work.
32              Potentially slow operation.
33              Normally should be performed only once.
34          """
35          self.username = settings.LABYRINTTI_USERNAME
36          self.password = settings.LABYRINTTI_PASSWORD
37          self.url = settings.LABYRINTTI_URL
```

Koodiesimerkki 12 Labyrintti Adapter-luokan startup-metodi

Kun MessagingService-luokasta on luotu instanssi, niin __init__-metodin toiminnot suoritettiin automaattisesti, mutta luokalle tehtiin vielä erikseen koodiesimerkissä 11 näkyvä startup-metodi. Kyseessä on tavallaan toinen alustus-metodi, mutta tällä metodilla ei ole Python tulkille erikoismerkitystä, kuten __init__-metodilla. Rivillä 111, kutsutaan backend-muuttujasta löytyvän Adapter-luokan instanssin samannimistä

metodia, joka tekee alustus toimintoja koodiesimerkin 12 mukaisesti Adapter-luokalle. MessagingService-luokan startup-metodin toimintoja ei haluttu laittaa __init__-metodiin, koska kyseiset toiminnot saattoivat olla raskaita ja viedä aikaa, joten rajapinta pitää vielä ”käynnistää” startup-metodilla. Koodiesimerkin 11 rivillä 112, started-muuttujalle asetetaan arvo True, joka kertoo, että rajapinta on käynnistetty.

Olen ottanut koodiesimerkkiin 12 esimerkkinä Labyrintti-SMS-palveluntarjoajaa varten luodun Adapter-luokan ja sen startup-metodin. Metodissa määritellään Labyrintti operaattori kohtaisia asetuksia, jotka taas haetaan palvelimen asetuksista. Näillä määrityksillä Adapter-luokka yhdistää internetin yli Labyrintin-SMS-yhdyskäytävään. Koska kyseessä on pelkästään Labyrintti-operaattoria varten luotu luokka, niin periaatteessa tiedot voitaisiin kirjoittaa suoraan koodiin, eikä hakea asetuksista, mutta tämä on huono käytäntö, koska jos esimerkiksi tulevaisuudessa tietoja pitää vaihtaa, niin on sen tekeminen kätevämpää asetuksista käsin, eikä tunnustietojen kirjoittaminen suoraan koodiin ole ikinä järkevää tietoturvasyistä. Koodiesimerkissä 12 voisi näkyä nyt Keyprolle luodut Labyrintti-tunnukset, mutta onneksi kirjoittaja on ollut tietoturvallinen. [24.] [25.]

5.3.2 Tekstiviestipyynnön eteneminen rajapinnassa

Kun rajapinta on alustettu ja käynnistetty koodiesimerkin 11 mukaisesti, niin rajapinta on valmis lähettämään tekstiviestejä. Tekstiviestin lähetykseen tarvitaan kuitenkin lähetystietoja, jotka annetaan rajapinnalle SmsParameters-luokasta luodun instanssin avulla, kuten aiemmissa luvuissa on todettu. Koodiesimerkissä 13 rivillä 165 luodaan ensiksi instanssi, jonka jälkeen instanssin luokkamuuttujille annetaan lähetykseen tarvittavia tietoja, kuten vastaanottajien puhelinnumeroita ja niin edelleen. Lähetystiedot saadaan asiakaspuolelta näkymään saapuneen HTTP-pyynnön kautta. HTTP-pyynnön eri parametreihin pääsee käsiksi Djangon HttpRequest-instanssin kautta, kuten riveillä 166-170 tehdään request-muuttujan avulla, joka on HttpRequest instanssi. Rivillä 173 lähetetään tekstiviesti kutsumalla rajapinnan send_sms-metodia ja argumenttina annetaan sms-muuttuja. [5, s.34.]

```
164         # Here we define the SMS parameters
165         sms = SmsParameters()
166         sms.message = request.POST['message']
167         sms.recipients = request.POST['recipients']
168         sms.sender = request.POST['sender']
169         sms.message_origin = request.POST['message_origin']
170         sms.request_by = request.POST['request_by']
171
172         #Here the parameters are sent to the API
173         sms_response = service.send_sms(sms)
```

Koodiesimerkki 13 Lähetystietojen määrittely ja tekstiviestin lähetys Django-näkymässä

```
115     def send_sms(self, sms_request):
116         """
117             Sends a text-message.
118         """
119         # Validation
120         ...
121         if not sms_request.recipients:
122             raise ValueError("Recipients cannot be empty")
123         ...
124
125         logger.debug("Saving the message to the database...")
126         sms = MessageRequestLog(
127                                     request_status="Sending",
128                                     request_response="Waiting response from the gateway ...",
129                                     text_message=sms_request.message,
130                                     message_recipients=sms_request.recipients,
131                                     message_sender=sms_request.sender,
132                                     request_by=sms_request.request_by,
133                                     message_origin=sms_request.message_origin,
134                                     message_type=sms_request.sms_type,
135         )
136         sms.full_clean()
137         sms.save()
138         logger.debug("Message saved")
139         logger.debug("Sending message to SMS Gateway")
140         ...
141         try:
142             adapter_response = self.backend.simple_send_sms(sms_request)
143         ...
144
145         except Exception as e:
146             ...
147             adapter_response.status = AdapterResponse.STATUS_ERROR
148             fatal_exception = e
```

Koodiesimerkki 14 MessagingService-luokan send_sms-metodin osa

Koodiesimerkissä 14 näkyy osia send_sms-metodista. Koodiesimerkissä koodia on pätkitty, joka ilmenee ...-merkinnöillä, kuten rivillä 118. Send_sms-metodi suorittaa

saamalleen SmsParameters instanssille eli sms_request-muuttujalle muutaman yksinkertaisen varmennuksen, kuten tarkastaa, että viestin vastaanottajan numero on annettu. Varmennuksen jälkeen instanssin muuttujien arvot tallennetaan tietokantaan käyttäen MessageRequestLog-luokkaa eli rajapinnan mallia.

MessaRequestLog-luokasta luodaan instanssi koodiesimerkin 14 rivillä 126 ja sille annetaan arvoja suoraan luonnin yhteydessä. Instanssin olisi voinut luoda samanlailla, kuten SmsParameters instanssi luotiin koodiesimerkissä 13, mutta kummatkin tavat ovat yhtä päteviä ja kyseessä on enemmänkin makuasia. MessageRequestLog-luokan instanssi on oma rajapintansa MessageRequestLog-taulun tietokantaoperaatioille. Vaikka MessageRequestLog-luokasta on nyt luotu instanssi ja sille on annettu alustavat arvot, niin Django ei ole vielä tehnyt yhtään varsinaista tietokantaoperaatiota eli mitään ei vielä ole tallennettu tietokantaan, kuten kappaleessa 3.4 käy ilmi. Rivillä 136 full_clean-funktio suorittaa muutaman Django-kohtaisen varmennuksen ja seuraavalla rivillä kutsutaan kaikilta Django-malleilta löytyvää save-metodia, jolloin Django tallentaa instanssin tiedot tietokantaan MessageRequestLog-tauluun. Viimeiseksi koodiesimerkissä 14 kutsutaan Adapter-luokan simple_send_sms-metodia, joka on nimensä mukaisesti yksinkertaistettu ja täysin operaattori kohtainen versio send_sms-metodista. [5, s. 84; 26.]

Simple_send_sms-metodi on kääritty Python Try - lauseen sisälle eli ohjelma yrittää suorittaa Try-lausekkeen sisällä olevan koodin, mutta jos jokin koodista aiheuttaa Python Exceptionin eli poikkeuksen, niin ohjelman suoritus siirtyy Exception-lauseen sisään, jossa reagoidaan poikkeukseen halutulla tavalla. Rivillä 145 ”as e”-kohta tuo e-muuttujan käyttöön Exception-lauseeseen. Muuttujassa on arvona poikkeuksen virheilmoitus. Minulle opetettiin, että Python poikkeuksia kannattaa ”ottaa kiinni” vain silloin, kun tiedetään tarkasti, mistä poikkeus johtuu ja että ohjelman on vielä järkevää jatkaa toimintaansa eli on myös täysin hyväksyttävää ja kannattavaa antaa ohjelman kaatua. Send_sms-metodi ei ole täysin malliesimerkki poikkeuksien kiinniottamisessa, koska se ottaa kiinni kaikki simple_send_sms-metodin aiheuttamat poikkeukset, joten ei ole mahdollista määrittää tarkasti, mistä poikkeus johtuu. Exception-luokka on yleinen poikkeus luokka, josta kaikki tarkemmin määritellyt poikkeus-luokat periytyvät, joten ohjelman tulisi mieluummin ottaa kiinni näitä tarkempia poikkeus-luokkia.

Esimerkiksi ZeroDivisionError-poikkeus ilmenee, jos Try-lauseen sisällä yritetään jakaa nollalla.

```
40    def simple_send_sms(self, parameters):
41        """  Sends a message to Labyrintti SMS gateway. Parameters are received from the services module
42         Parameters are actually an instance of a SmsMessageRequest found in keycore_messaging.services module
43         ...
44
45        """
46        adapter_resp = AdapterResponse()
47        ....
48
49        #SMS parameters put into a HTTP request
50        params = {
51                "user":self.username,
52                "password":self.password,
53                "source-name":str(parameters.sender),
54                "dest":dest,
55                "class":parameters.sms_type,
56                "text":parameters.message.encode('iso-8859-1'),
57        }
58
59        data = urllib.urlencode(params)
60        # Connection attempt to the Labyrintti gateway
61        req = urllib2.Request(self.url, data)
62        # Connects to GW
63        http_response = urllib2.urlopen(req)
64        # Reads the HTTP Response as text, which is going to be saved in the database
65        adapter_resp.service_response = http_response.read()
66        ...
67
68        adapter_resp.recipient_status = self.parse_http_response(adapter_resp.service_response)
69        logger.debug("Parsing done")
70        # status can be STATUS_OK or STATUS_ERROR. This concerns only about whether connection was succesful to gateway.
71        adapter_resp.status = AdapterResponse.STATUS_OK
72
73        return adapter_resp
```

Koodiesimerkki 15 Ote Labyrintti Adapter-luokan simple_send_sms-metodista

Adapter-luokka haluttiin pitää mahdollisimman yksinkertaisena ja jokaisen Adapter-luokan simple_send_sms-metodin tehtävänä oli muuntaa parametrina saadun SmsParameters-instanssin lähetystiedot oikeaan muotoon, lähettää tiedot internetin yli operaattorille ja lähetyksen onnistumisesta riippuen palauttaa vastauksena AdapterResponse-luokasta muodostettu vastaus (ks. liite 2).

Koodiesimerkki 15 on otos Labyrintti Adapter-luokan simple_send_sms-metodista. Metodi aloittaa luomalla AdapterResponse-luokan instanssin, jota käytetään myöhemmin metodin palautusarvona, jos yhteys operaattoriin onnistuu. Sitten on kuvasta on pätkitty muutama Labyrintti-kohtainen varmennus ja seuraavaksi rivillä 50

luodaan params-muutuja, joka saa arvoksi Python dictionaryn, jolle on arvoksi annettu tekstiviestin lähetystietojen lisäksi lisäparametreja, joiden avulla urllib2-kirjasto muodostaa yhteyden Labyrintti operaattoriin internetin yli. Rivillä 59 aikaisempi params-muuttuja koodataan HTTP-protokollan mukaiseen muotoon. Rivillä 61 luodaan urllib2-kirjastosta löytyvän Request-luokan instanssin, joka ilmentää HTTP-pakettia. Sitten paketti lähetetään internetin ihmeelliseen maailmaan urlopen-metodin avulla ja kuunnellaan, mitä operaattori vastaa. Jos operaattori ei jostain syystä vastaa, niin urllib2 laukaisee Python-poikkeuksen, joka siirtää rajapinnan suorituksen takaisin send_sms-metodiin, jossa poikkeukseen reagoidaan Exception-lauseessa. Oletetaan, että kaikki menee hyvin ja operaattori vastaa, niin muunnetaan vastaus read-metodin avulla merkkijono-muotoon, joka lähetetään edelleen parse_http_response-metodille rivillä 68. [27.] [28.].

parse_http_response-metodi parsii saamansa merkkijonon käyttäen säännöllistä lauseketta ja selvittää parsitusta vastauksesta, onnistuiko tekstiviestin lähetys vai ei, ja muodostaa AdapterResponse-luokan mukaisen vastauksen.

Rivillä 71 status-luokkamuuttujaan päivitetään, että yhteys onnistui operaattorille ja sitten metodi palauttaa AdapterResponse-luokan instanssin, josta käy ilmi, kuinka tekstiviestin lähetys sujui operaattorin päädyssä.

```
140         ...
141         try:
142             adapter_response = self.backend.simple_send_sms(sms_request)
143         ...
144
145         except Exception as e:
146             ...
147             adapter_response.status = AdapterResponse.STATUS_ERROR
148             fatal_exception = e
149
150         # Updates the SMS sending status using the response from the adapter
151         self.update_sms_status(sms, adapter_response)
152
153         # 0 = "OK"
154         if adapter_response.status == adapter_response.STATUS_OK:
155
156             #Returns the status code (0=success, -1=Failure), and the id to identify the SMS
157             return {'code':0, 'message_id':sms.request_id}
158
159         else:
160             return {'code':1, 'message_id':sms.request_id, 'exception':fatal_exception}
```

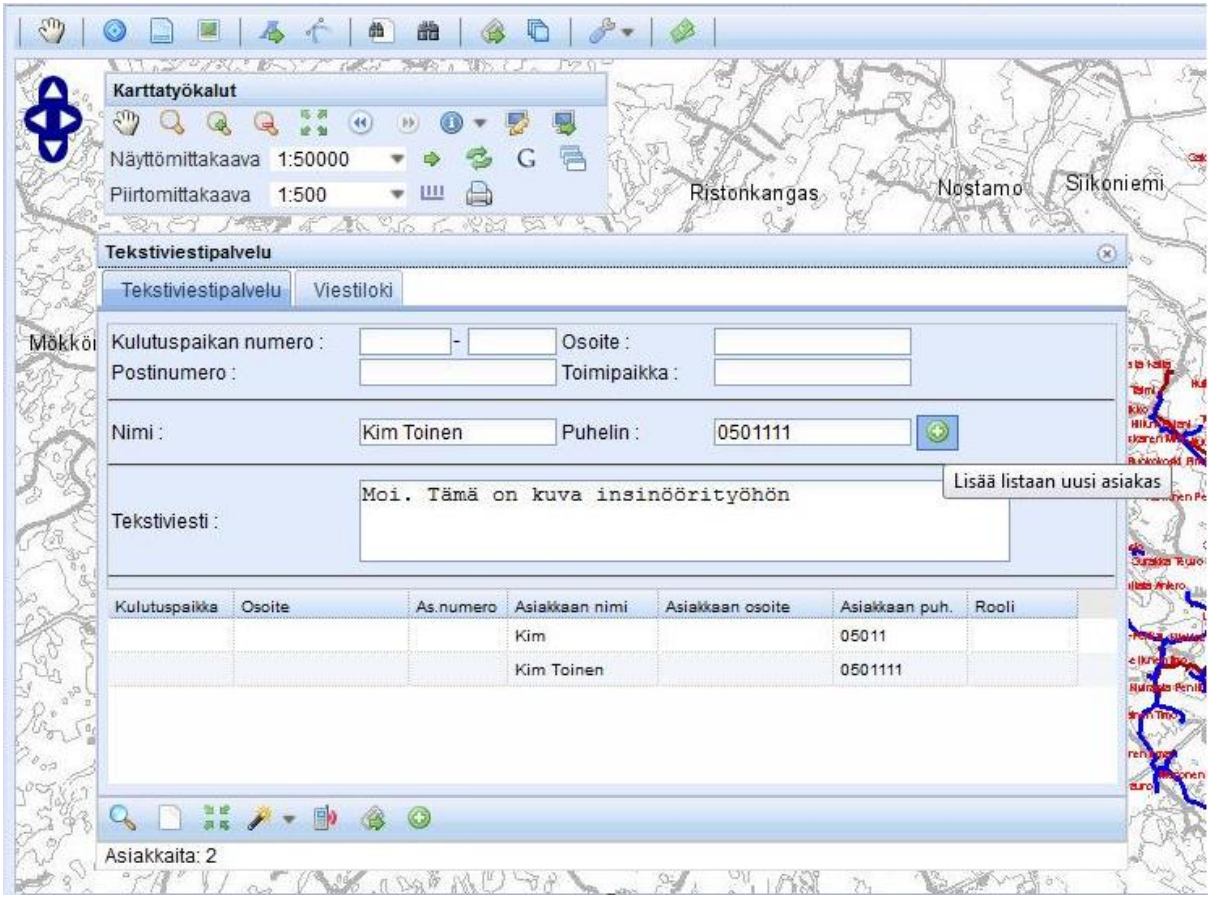
Koodiesimerkki 16 Otos MessagingService-luokan send_sms-metodista

Ohjelman suoritus siirtyy takaisin send_sms-metodiin, joka on nähtävissä koodiesimerkissä 16, joka jatkaa siitä mihin koodiesimerkki 15 jäi. Rivillä 151 kutsutaan update_sms_status, jolla annetaan argumentteina aikaisemmin luotu MessageRequestLog instanssi ja Adapter-luokalta vastauksena saatu AdapterResponse-luokan instanssi. Update_sms_status päivittää tietokantaan tekstiviestin lähetyksen tilan, jonka jälkeen rajapinta vastaa sitä alunperin kutsuneelle näkymälle, miten tekstiviestin lähetys onnistui rivien 157 tai 160 mukaisesti. Näkymän tehtävänä on tulkata vastaus ja lähettää tulos edelleen asiakaspuolelle, jossa loppukäyttäjälle ilmoitetaan, onnistuiko tekstiviestin lähetys.

5.4 Asiakaspuoli eli rajapinnan visuaalinen osa

Tämän työn tekstiviestirajapinta toimii palvelinpuolella, mutta rajapinta on tarkoitettu ensisijaisesti asiakkaiden käyttöön tekstiviestien lähetykseen, joten se tarvitsee myös visuaalisen osan asiakaspuolelle. Haluan korostaa, etten ohjelmoinut tätä osuutta itse, mutta autoin lomakkeen luonnissa muun muassa kertomalla, mitä kenttiä se tarvitsee kommunikoinnissa rajapinnan kanssa. Haluan lyhyesti esitellä tämänkin osan, jotta lukija saa kokonaiskuvan rajapinnan toimivuudesta.

Web-sovelluksissa olennaista osaa näyttelevät HTML-lomakkeet, joita käytetään asiakaspuolen ja palvelimen väliseen kommunikaatioon. KeyCore-tuotteet eivät tee tässä asiassa poikkeusta, sillä kaikki toiminnot, kuten uuden kaapelin luominen, putken poistaminen tai teletilan tietojen muokkaaminen tapahtuu HTML-lomakkeen kautta. Myös tekstiviestirajapinnan käyttämiseen luotiin kuvan 4 mukainen lomake. Lomakkeen kenttien avulla käyttäjä antaa tarvittavat perustiedot tekstiviestin lähetykseen ja painaa ”Lähetä tekstiviesti”-nappia, joka löytyy lomakkeen alapalkissa kännykkä-ikonin takaa. Lomake lähettää HTTP-pyynnön palvelimelle, joka ohjaa pyynnön tekstiviestirajapinnalle, joka sitten hoitaa tekstiviestin lähetyksen ja palauttaa HTTP-vastauksen, josta käy ilmi onnistuiko tekstiviestin lähetys. Tämä vastaus esitetään käyttäjälle lomakkeen alalaidassa. [5, s. 119.]



Kuva 4 Tekstiviesti-lomake avattuna KeyAquassa

6 Testaus

6.1 Motivaatio testaukseen

Aikaisemmin käsitellyistä hyvistä ohjelmointitavoista testaus on mielestäni tärkeimpiä ohjelman toimivuuden ja ohjelmoijan mielenterveyden puolesta. Itse opin tämän kantapään kautta ja huomasin usein kirjoittavani testejä vasta usean ohjelmointipäivän jälkeen, kun ohjelma ei toiminut enää jonkin muutoksen jälkeen ja joudun rivi riviltä etsimään virhettä testien avulla. Ohjelmoijan tehdessä muutoksia aiempaan koodinsa, niin on riski, että se rikkoo ohjelman osan, joka toimi aiemmin. Pahimmassa tapauksessa ohjelmoijalla kestää erittäin kauan jäljittää vika siihen ohjelman osaan, joka on toiminut aiemmin. Jatkuvasti testejä kirjoittamalla ja ajamalla samat testit aina uudestaan läpi ohjelmoija huomaa paljon helpommin, jos ohjelma on päässyt

rikkoutumaan. On totta, että testien kirjoittaminen normaalin ohjelmoinnin lisäksi voi olla puuduttavaa, mutta Django tekee siitä helppoa ja kannattavaa. [12.]

```
487         vastaus = 1+1
488         self.assertEqual(vastaus, 2)
```

Koodiesimerkki 17 Esimerkki väittämä, että "vastaus-muuttuja on sama kuin numero 2"

Python tarjoaa kaksi päätestaustyyliä: Doctests ja Unittests. Minua neuvottiin käyttämään Unittests-tapaa ja Djangon dokumentin mukaan sitä käyttävät myös kokeneimmat Web-kehittäjät. Djangon Unittests pohjautuu Python unittest-kirjastoon ja testaus perustuu eräänlaisiin väittämiin (assertions). Oletusasetuksilla ohjelmoijan täytyy kirjoittaa Django-sovelluksen juureen tests.py, josta Django osaa itsestään ajaa testit. Kuten koodiesimerkissä 18 näkyy, ohjelmoija kirjoittaa niin sanottuja testiluokkia, joissa hän esittää erilaisia väittämiä vasten testattavaa lähdekoodia ja tällä tavalla käy läpi, että testattava sovellus toimii niin kuin pitää. Esimerkiksi koodiesimerkissä 17 ohjelmoija väittää, että vastaus-muuttujan arvo on 2 ja tämä on totta, joten testi menee läpi onnistuneesti. Testin epäonnistuessa ohjelma kaatuu ja näyttää, mikä testeistä epäonnistui. Ohjelmoija näkee suoraan, miten ohjelman olisi pitänyt toimia, joka helpottaa suuresti ongelman jäljittämisessä ja korjaamisessa. [12.]

Doctest-tapaa ei tässä projektissa käytetty, mutta kyseessä on samanlainen testaustyyli kuin Unittest, jossa esitetään väittämiä, ja jos jokin väittämä ei pidä paikkaansa, niin testi epäonnistuu. Doctest-testit kirjoitetaan kuitenkin metodin docstring-kohtaan, mikä mielestäni sotkee koodin luettavuutta, jos testit kasvavat suureksi. [13.]

Djangon mukana tulee myös erittäin kätevä kehitykseen tarkoitettu web-palvelin, jonka avulla voi mallintaa oikeata web-palvelinta. Kyseisen palvelimen pystyin ajamaan Keycore tuotteita omalla työasemallani yhden rivin pituisella komentorivin komennolla: **python manage.py runserver** . Näin kehittäjä pystyy välttämään erillisen web-palvelimen konfiguroinnin ja hallinnoinnin, ja vaikka Djangon palvelin on näinkin kevyt, niin se suoriutuu erinomaisesti kookkaan verkkotietojärjestelmän pyörityksessä ja toimi olennaisena osana testauksessa. [5, s. 18-19.]

6.2 Unit test-testaus

Tässä kappaleessa käydään lyhyesti läpi yksi osa Labyrintin testiluokasta ja selostetaan, kuinka unit test-testaus toimii Djangossa. Testiluokan test_sms_send_real-metodi nimensä mukaisesti testaa oikeata viestin lähetystä. Alempana koodissa tulisi vastaan lukuisia metodeja, jotka testaavat eri osia sovelluksesta. Sääntönä oli, että yksi metodi jokaista testitapausta kohti ja metodien nimien tuli kuvata tarkasti mitä testataan. Eli aina kun kirjoitin uuden osan sovellukseen, niin kirjoitin myös testimetodin kyseiselle osalle.

```
143 class LabyrinttiMessagingServiceTestCase(TestCase):
144     def test_sms_send_real(self):
145         """
146             Real integration testing using real Labyrintti adapter.
147             Should be normally disabled, see settings LABYRINTTI_INTEGRATION_TEST_ENABLED
148         """
149         if not settings.LABYRINTTI_INTEGRATION_TEST_ENABLED:
150             logger.warning("Labyrintti real integration testing is disabled, see settings LABYRINTTI_INTEGRATION_TEST_ENABLED")
151             return
152         logger.warning("Labyrintti real integration testing is ENABLED!")
153 
154         #use own, isolated service instance
155         service = MessagingService(settings.KEYCORE_MESSAGING_BACKEND_LABYRINTTI)
156         service.startup()
157 
158         #Override params using real values for the integration test
159         service.backend.url = settings.LABYRINTTI_URL_REAL
160         service.backend.username = settings.LABYRINTTI_USERNAME_REAL
161         service.backend.password = settings.LABYRINTTI_PASSWORD_REAL
162 
163 
164         # Here we define the SMS parameters
165         sms = SmsMessageRequest()
166         #use "hardcoded" data here as we will compare results
167         #in the end and should not depend on external settings for that
168         sms.message = u'Öylätti'
169         sms.recipients = settings.LABYRINTTI_PHONES_REAL
170         sms.sender = u'Keypro'
171         sms.message_origin = 'KeyCore'
172         sms.request_by = 'Tester'
173 
174         #Here the parameters are sent to the API
175         sms_response = service.send_sms(sms)
176         self.assertIsNotNone(sms_response, "Successful service response must not be None")
```

Koodiesimerkki 18 Osa testiluokan test_send_sms_real-metodista

Koodiesimerkissä 18 riveillä 149-152 on ehtolause, joka helpottaa testauksen vaihtamista aidon ja tekaistun tekstiviestin lähetyksen välillä, mutta lauseella ei ole

olennaista merkitystä testauksen kannalta. Riveillä 155-175 tehdään aiemmin käsitellyt rajapinnan alustustoiminnot. Rivillä 176 tehdään ensimmäinen väittämä eli koodissa väitetään, ettei sms_response ole None-tyyppi ja jos näin on, niin väittämä on läpäisty. Jos sms_response sattuu olemaan None, niin testien ajo pysähtyy ja tulostetaan virheilmoitus: ”Successful service response must not be None”. Näin virheen pystyy jäljittämään tarkkaan ja virhe tekstistä käy vielä ilmi, miten testattavan sovelluksen ei tulisi toimia. Yhteensä erilaisia väittämiä rajapinnan kohdalle kasaantui noin 50, joiden avulla kehitys pysyi hyvin koossa.

Testien kirjoituksella on myös ei niin näkyvä positiivinen vaikutus nimittäin se pakottaa ohjelmoijan ajattelemaan ”kuinka tämän ohjelman tulisi toimia tässä kohtaan”. Itse usein lähdin kirjoittamaan uutta osaa sovellukseen sen tarkemmin ajattelematta, jolloin ohjelman saattoi toimia epäloogisesti, mutta testien kirjoitusten takia jouduin usein pysähtyä tarkemmin miettimään ohjelman toimivuutta.

Tekstiviestirajapinta saatiin valmiiksi, se toimi testauksissa niin kuin toivottiin, ja lopputulokseen oltiin tyytyväisiä. Käyttöönotto ei itsessään ollut mikään suuri rituaali, koska kyseessä on erikseen laskutettava ominaisuus tuotteessa, niin tekstiviestitoiminto otetaan erikseen käyttöön asiakaskohtaisesti. Testaukset kuitenkin tehtiin tuotanto ympäristössä, joten tekstiviestirajapinta on valmis käyttöönotettavaksi.

7 Tekstiviestirajapinnan käyttökohteita

7.1 Kysyntä tekstiviestitoiminnolle

Tekstiviestipalvelut ovat todella yleisiä, ja vaikka kyseisiä palveluita on ollut saatavilla jo jonkin aikaa, niin ilmiö tuntuisi olevan kasvamaan päin. Minun ensimmäisiä kokemuksia tekstiviestipalveluista oli limun tilaus tekstiviestillä yläasteen limukoneesta, sitten pari vuotta myöhemmin tekstiviestilippu metroon ja juuri viime viikonloppuna tein lähtöselvityksen lentokoneeseen ensimmäistä kertaa tekstiviestillä. Myös Keypro ja yhteiskehityksessä mukana olevat asiakkaat näkivät potentiaalin tekstiviestissä.

Tekstiviestien lähettäminen suoraan selaimesta huvikseen voi kuulostaa hauskalta, mutta kyseinen ominaisuus tuo mukanaan syvällisempiä mahdollisuuksia, kuten varoitus tekstiviestillä tai tekstiviestiin perustuvan autentikoinnin.

7.2 Asiakkaiden varoittaminen massatekstiviestillä

Keypron asiakaskuntaan kuuluu muun muassa useampi vesiosuuskunta. Vesiosuuskunnat käyttävät luonnollisesti viemäröinti- ja putkistoverkkojen suunnitteluun soveltuvaa KeyAqua-tuotetta. Yhtenä toivomuksena oli tuki massatekstiviestien lähetykseen varoitus tarkoituksessa suoraan KeyAquasta. Kuten johdannossa mainittiin, niin varoitus voi esimerkiksi olla saastuneesta vedestä tietyllä alueella ja tekstiviesti on tehokas tapa tavoittaa kuka vain. [14.]

En itse testannut tekstiviestin lähetystä viittä henkilöä isommalle ryhmälle, joten en osaa yhtään sanoa, miten järjestelmä reagoi massatekstiviestiin esimerkiksi 1000 asiakkaalle. Yksittäinen viesti kuitenkin kulkee selaimesta palvelimelle ja palvelimelta operaattorille eli selaimessa toimiva sovellus voi jumittua täysin tai palvelin ei kykene vastaanottamaan sellaista määrä viestejä samanaikaisesti, tai palvelin suoriutuu, mutta operaattori ei. Kaikki tämä on kuitenkin testien puutteesta johtuvaa spekulaatiota ja järjestelmää tullaan testaamaan suuremmalla joukolla, kun aika koittaa.

7.3 Tekstiviestiin perustuva autentikointi

Tekstiviestiautentikointi toisi lisäturvallisuutta Keypron tuotteille. Kun tekstiviestiautentikointi on käytössä, niin käyttäjä sisäänkirjautuu ensiksi normaalilla tavalla ja jos kirjautuminen onnistuu, niin asiakkaan puhelimeen saapuu tekstiviestillä kertakäyttösalasana, jonka asiakkaan tulee syöttää kirjautuakseen sisään. Tällä tavoin jos asiakas sattuu hävittämään tunnuksensa, niin niitä ei voi väärinkäyttää yksistään. Samantyylistä tekstiviestiin perustuvaa lisäautentikointia käyttää verkkopalvelussaan muun muassa OP. [31.]

7.4 Näytön ajankohta

Näytön ajankohdan ilmoittaminen kaivajille tekstiviestillä oli yksi jo tiedossa oleva käyttökohde rajapinnalle. Keypron asiakkaina on myös johdonomistajia, joilta kaivajat tilaavat näyttöjä, jotta tiedetään missä johdot kulkevat. Johdonomistajat suunnittelivat käyttävänsä tekstiviestiä näytön ajankohdan ilmoittamiseen kaivajalle. [14, 15.]

8 Jatkokehitys

Opin että ohjelmoija näkee työssään aina parannettavaa ja tuntui, ettei aika koskaan riittänyt eri projekteissa, mutta muutama isompi parannusehdotus jäi ilmoille tekstiviestirajapinnan osalta.

Edellisessä luvussa kirjoitin massatekstiviestien käytöstä varoitusmielessä. Mielestäni tämä on hyödyllinen ominaisuus ja tämä kehitysehdotus ei suoranaisesti koske itse tekstiviestirajapintaa, vaan sen asiakaspuolta, jolta viestit lähetetään. Tämän lomakkeen pitäisi tukea laajoja asiakasrekistereitä ja asiakkaiden jakoa asiakasryhmiin, jotta massaviestitystä voisi käyttää järkevästi.

Ilmoitusviesti siitä, että tekstiviesti on mennyt perille asti on myös tärkeä ominaisuus. Tässä tapauksessa tekstiviestirajapintaa pitää muokata niin, että se osaa ottaa vastaan näitä ilmoituksia. Esimerkiksi varoitustilanteissa on tärkeää tietää, jos viesti ei koskaan saapunut perille vastaanottajan puhelimeen, jolloin voidaan tehdä jatkotoimenpiteitä tai jos viesti ei koskaan tavoittanut kaivajaa, niin hänelle ilmoitetaan ajankohta muuta kautta.

Tekstiviestien vastaanotto rajapinnassa on myös toteutettavissa samalla periaatteella, kuin yllä mainittu ilmoitusviesti. Tekstiviestien vastaanotto ja järjestelmän reagointi vastaanotettuihin viesteihin voi avata tietä innovatiivisille ominaisuuksille. Niin kauan kuin käytetään kolmansia osapuolia tekstiviestien välitykseen, niin niiden pitää myös tukea viestien vastaanottoa, mutta ehkä tulevaisuudessa Keypro Oy siirtyy omaan viestipalvelimeen.

9 Lopetus

Itse suuntauduin koulussa tietoverkkoihin ohjelmoinnin tai tietoliikenteen sijaan, joten minulla oli todennäköisesti alussa enemmän opeteltavaa kuin vastaavassa tilanteessa ohjelmointiin suuntautuneella opiskelijalla olisi ollut. Ohjelmointitaitoni kehittyivät kuitenkin huimasti jo muutamassa kuukaudessa ja opinnäytetyön päätavoite saavutettiin eli KeyCoressa toimiva rajapinta tekstiviestien lähetykseen. Suurimmat ongelmat tekstiviestirajapinnan toteutuksen aikana johtuivat ihan Django-tietämyksen puutteesta, jota pohjusti myös vähäinen ohjelmointikokemus.

Tekstiviestirajapinta ei ollut ohjelmoinnin osalta loppujen lopuksi hirveän monimutkainen, mutta kuten tästä opinnäytetyöstä käy ilmi, niin suunnittelulla on todella iso osa koko prosessissa ja jopa ohjelmoinnin kannalta yksinkertaisissa sovelluksissa kannattaa panostaa suunnitteluun, jolloin itse ohjelmointivaihe selkenee ja vähentää muun muassa virheitä. Suunnitelmassa voi käyttää pohjana MVC:n tyylistä arkkitehtuuria, jonka kautta saavutetaan sovelluksen modulaarisuutta, jonka edut kävivät ilmi tekstiviestirajapinnan kohdalla. Hyvä suunnittelu on itsessään oma taiteenlajinsa, joka vaatii aika perusteellisen tietämyksen ympäristöstä ja käytettävästä kielestä, mutta jopa yksinkertainen suunnitelma auttaa jo paljon, ja voi suunnitelmaa parannalle toteutuksen aikanakin, kuten rajapinnan kohdalla tehtiin.

Kyseessä on tuore ominaisuus Keypro Oy:n verkkotietojärjestelmässä, joten rajapinnan täyttä potentiaalia ei ole vielä nähty, mutta tekstiviestille voi keksiä yllättävän monia käyttötarkoituksia ajan mittaan nyt kun rajapinta siihen on olemassa kasvavassa web-pohjaisessa verkkotietojärjestelmässä.

Lähteet

1. Keypron toimiala. 2012. Verkkodokumentti. <http://www.keypro.fi/fi/services>. Luettu 2012.

2. Uusitalo, Kirmo. 2012. Teknologiajohtaja. Keypro. KeyCore PowerPoint esitys. Sähköinen dokumentti. <Core-TeroAnttonen-KirmoUusitalo.ppt>

3. Paikkatietojärjestelmä. 2012. Verkkodokumentti. <http://fi.wikipedia.org/wiki/Paikkatietoj%C3%A4rjestelm%C3%A4>. Luettu 2012.

4. Verkkotietojärjestelmän esittely. 2012. Verkkodokumentti. <http://www.keypro.fi/fi/verkkotietoj%C3%A4rjestelm%C3%A4> . Luettu 2012.

5. Holovaty, Adrian ja Kaplan-Moss, Jacob. 2009. The Definitive Guide to Django – Web Development Done Right – Second edition. New York: Apress.

6. Tietojenkäsittelylaitos. Suunnitteludokumentti. Verkkodokumentti. <http://www.cs.helsinki.fi/u/tapasane/Ohjelmoinnin_harjoitustyo/suunnitteludokumentti.html>. Luettu 2012.

7. PyDev esittely. Verkkodokumentti. <http://pydev.org/>. Luettu 2012.

8. Eclipse-suunnitteluohjelmisto. Verkkodokumentti. <http://en.wikipedia.org/wiki/Eclipse_%28software%29>. Luettu 2012.

9. Mercurial-versionhallintajärjestelmä. Verkkotietojärjestelmä. <http://fi.wikipedia.org/wiki/Mercurial>. Luettu 2012.

10. Spolsky, Joel. Johdatus Mercurialiin. Verkkodokumentti. <http://hginit.com/>. Luettu 2012.

11. Django asetukset. Verkkodokumentti.
<https://docs.djangoproject.com/en/dev/topics/settings/>. Luettu 2012.

12. Testaus Djangossa. Verkkodokumentti.
<https://docs.djangoproject.com/en/dev/topics/testing/?from=olddocs>.
Luettu 2012.

13. Python doc-testaus. Verkkodokumentti.
<http://docs.python.org/library/doctest.html>. Luettu 2012.

14. Uusitalo, Kirmo. 2012. Teknologiajohtaja. Keypro, Helsinki. Yleinen keskustelu Keyprosta 2012.

15. Keypron johtoselvityspalvelu. Verkkodokumentti.
<http://www.keypro.fi/fi/johtoselvitys>. Luettu 2012.

16. Python-ohjelmointikieli. Verkkodokumentti.
<http://en.wikipedia.org/wiki/Python_%28programming_language%29>.
Luettu 2012.

17. Whetting your appetite. Verkkodokumentti.
http://docs.python.org/tutorial/appetite.html. Luettu 2012.

18. Kraft, Kim. Insinöörityöprojektin työpäiväkirja keväällä 2012.

19. Writing your first Django app, osa 3. Verkkodokumentti.
<https://docs.djangoproject.com/en/dev/intro/tutorial03/>. Luettu 2011.

20. Cross site scripting-tietoturva-aukko. Verkkodokumentti.
<http://fi.wikipedia.org/wiki/Cross_site_scripting>. Luettu 2012.

21. Valums, Andrew. Web apps vs desktop apps. 2010. Verkkodokumentti.
<http://valums.com/web-apps/>. Luettu 2012.

22. Modular programming. Verkkodokumentti. <http://en.wikipedia.org/wiki/Modular_programming>. Luettu 2012.

23. Python classes. Verkkodokumentti. <http://docs.python.org/2/tutorial/classes.html>. Luettu 2012.

24. Use of hard-coded Credentials. Verkkodokumetti. <http://minsky.gsi.dit.upm.es/semanticwiki/index.php/Category:Use_of_Hard-coded_Credentials>. Luettu 2012.

25. Extracting hard-coded credentials using managed code debugging techniques in Windbg. 2012. Verkkodokumentti. <http://www.exploit-monday.com/2012/05/extracting-hard-coded-credentials-using.html>. Luettu 2012.

26. Model instance reference. Verkkodokumentti. <https://docs.djangoproject.com/en/dev/ref/models/instances/?from=olddocs>. Luettu 2012.

27. urllib, Python-kirjasto. Verkkodokumentti. <http://docs.python.org/2/library/urllib.html>. Luettu 2012.

28. urllib2, Python-kirjasto. Verkkodokumentti. <http://docs.python.org/2/library/urllib2.html>. Luettu 2012.

29. Nations, Daniel. Web applications. Verkkodokumentti. <http://webtrends.about.com/od/webapplications/a/web_application.htm>. Luettu 2012.

30. Cross-platform. Verkkodokumentti. <http://en.wikipedia.org/wiki/Cross-platform>. Luettu 2012.

31. 2011. Avainluku ja maksun lisävahvistus lisäävät OP-verkkopalvelun turvallisuutta. Verkkodokumentti. <https://www.op.fi/op/henkiloasiakkaat/opastus/avainluku-ja-maksun-

lisavahvistus-lisaavat-op-verkkopalvelun-turvallisuutta?cid=151513742&srcpl=3>. Luettu 2012.

32. Keypron toiminta. Verkkodokumentti. <http://www.keypro.fi/fi/toiminta>. Luettu 2012.

33. Parameters and arguments. Verkkodokumentti. <http://en.wikipedia.org/wiki/Parameter_%28computer_programming%29#Parameters_and_arguments>. Luettu 2012.

34. What is the difference between a method and a function. Verkkodokumentti. <http://stackoverflow.com/questions/155609/what-is-the-difference-between-a-method-and-a-function>. Luettu 2012.

SmsParameters-luokka

```
14  class SmsParameters(object):
15      """ Define your SMS parameters here.
16
17          Message: "Your text message"
18          Recipient (Labyrintti)(tuple): Phone number(s) in a tuple
19          Sender (Labyrintti)(str/int): Sender's name or a number shown on receivers phone.
20                  (MAX 11 characters or 16 digits)
21          Sms_type (Labyrintti): "normal" or "flash"
22      """
23
24      def __init__(self):
25          #String, not empty, "send from" shown on the phone
26          self.sender = None
27          #String, not empty, name of the person (user) who requested this sms to be sent
28          self.request_by = None
29          #String, not empty, name of the service/product/installation requested this operation
30          self.message_origin = None
31          #Collection, not empty
32          self.recipients = []
33          #String, not empty, 'normal' or 'flash'
34          self.sms_type = u'normal'
35          #String, not empty, message to send, unicode
36          self.message = None
37
```

AdapterResponse-luokka

```
class AdapterResponse(object):
    # The raw response from the gateway
    SERVICE_RESPONSE = ""

    #accepted for sending
    STATUS_OK = 0

    #general error
    STATUS_ERROR = 1

    # If none of the errors below occurs then this variable is used to point success
    RECIPIENT_OK = 0

    #Unknown error: Error whose reason is not known or specified
    RECIPIENT_ERROR = 1

    #Invalid recipient: Recipient phone number syntax is invalid. For example, too short or long.
    RECIPIENT_ERROR_INVALID = 2

    #Duplicate recipient: The same recipient phone number has been specified multiple times. Only one message will be sent to each phone number.
    RECIPIENT_ERROR_DUPLICATE = 3

    #Unallowed recipient: Recipient phone number is not allowed in user account configuration. For example, foreign recipient phone numbers can be denied.
    RECIPIENT_ERROR_UNALLOWED = 4

    #Routing error: There is no operator route that supports sending to the recipient phone number.
    RECIPIENT_ERROR_ROUTING_ERROR = 5
```